Logtalk 2.6 Documentation

July 2000

Paulo Jorge Lopes de Moura
pmoura@noe.ubi.pt

TECHNICAL REPORT DMI—2000/1

Department of Mathematics and Informatics
University of Beira Interior
Rua Marquês d'Ávila e Bolama
6201-001 Covilhã — Portugal

Table of contents

User Manual

Logtalk features

Some years ago, I decided that the best way to learn object-oriented programming was to build my own object-oriented language. Prolog always being my favorite language, I chose to extend it with object-oriented capabilities. Eventually this work has lead to the Logtalk system. The first public release of Logtalk 1.x occurred in February of 1995. Based on feedback by users and on the author subsequent work, the second major version went public in July of 1998.

Although this version of Logtalk shares many ideas and goals with previous 1.x versions, programs written for one version are not compatible with the other (however, conversion from previous versions can easily be accomplished in most cases). This is a consequence of the desire to have a more friendly system, with a very smooth learning curve, bringing Logtalk programming closer to traditional Prolog programming. There are, of course, also other important changes, that result in a more powerful and funnier system. Logtalk 2.x development provides the following features:

Integration of logic and object-oriented programming

Logtalk tries to bring together the main advantages of these two programming paradigms. On one hand, the object orientation allows us to work with the same set of entities in the successive phases of application development, giving us a way of organizing and encapsulating the knowledge of each entity within a given domain. On the other hand, logic programming allows us to represent, in a declarative way, the knowledge we have of each entity. Together, these two advantages allow us to minimize the distance between an application and its problem domain, turning the writing and maintenance of programming easier and more productive.

In a more pragmatically view, Logtalk objects provide Prolog with the possibility of defining several namespaces, instead of the traditional Prolog single database, addressing some of the needs of large software projects.

Integration of event-driven and object-oriented programming

Event-driven programming enables the building of reactive systems, where computing which takes place at each moment is a result of the observation of occurring events. This integration complements object-oriented programming, in which each computing is initiated by the explicit sending of a message to an object. The user dynamically defines what events are to be observed and establishes monitors for these events. This is especially useful when representing relationships between objects that imply constraints in the state of participating objects [Rumbaugh 87, Rumbaugh 88, Fornarino 89, Razek 92]. Other common uses are reflective applications like code debugging or profiling [Maes 87].

Support for both prototype and class-based systems

Almost all (if not all) object-oriented languages available today are either class-based or prototype-based [Lieberman 86], with a strong predominance of class-based languages. Logtalk provides support for both hierarchy types. That is, we can have both prototype and class hierarchies in the same application. Prototypes solve a problem of class-based systems where we sometimes have to define a class that will have only one instance in order to reuse a piece of code. Classes solves a dual problem in prototype based systems where it is not possible to encapsulate some code to be reused by other objects but not by the encapsulating object.

Stand-alone objects, that is, objects that do not belong to any hierarchy, are a convenient solution to encapsulate code that will be reused by several unrelated objects.

Support for multiple object hierarchies

Languages like Smalltalk-80 [Goldberg 83], Objective-C [Cox 86] and Java [Gosling et al. 96] define a single hierarchy rooted in a class usually named `Object`. This makes it easy to ensure that all objects share a common behavior but also tends to result in lengthy hierarchies where it is difficult to express objects that represent exceptions to default behavior. In Logtalk we can have multiple, independent, object hierarchies. Some of them can be prototype-based while others can be class-based. Furthermore, stand-alone objects provide a simple way to encapsulate utility predicates that do not need or fit in an object hierarchy.

Separation between interface and implementation

This is an expected (should we say standard?) feature of almost any modern programming language. Logtalk provides support for separating interface from implementation in a flexible way: protocol directives can be contained in an object, a category or a protocol (first-order entities in Logtalk) or can be spread in objects, categories and protocols.

Private, protected and public inheritance

Logtalk supports private, protected and public inheritance in a similar way to C++ [Stroustrup 86], enabling us to restrict the scope of inherited, imported or implemented predicates (by default inheritance is public).

Private, protected and public object predicates

Logtalk supports data hiding by implementing private, protected and public object predicates in a way similar to C++ [Stroustrup 86]. Private predicates can only be called from the container object. Protected predicates can be called by the container object or by the container descendants. Public predicates can be called from any object.

Parametric objects

Object names can be compound terms (instead of atoms), providing a way to parameterize object predicates. Parametric objects are implemented in a similar way to `L&O` [McCabe 92], `OL(P)` [Fromherz 93] or `SICStus Objects` [SICStus 95]. However, access to parameter values is done via a built-in method instead of making the parameters scope global over the whole object.

Smooth learning curve

Logtalk has a smooth learning curve, by adopting standard Prolog syntax (using a pre-processor) and by enabling an incremental learning and use of most of its features.

Compatibility with most Prologs and the ISO standard

The Logtalk system has been designed to be compatible with most Prolog compilers and, in particular, with the ISO Prolog standard [ISO 95]. It runs in almost any computer system with a modern Prolog compiler.

Message sending

Note that message sending is only the same as calling an object's predicate if the object does not inherit (or import) predicate definitions from other objects (or categories). Otherwise, the predicate definition that will be executed may depend on the relationships of the object with other Logtalk entities.

Operators used in message sending

Logtalk uses the following three operators for message sending:

```
:- op(600, xfx, ::).
:- op(600,  fx, ::).
:- op(600,  fx, ^^).
```

It is assumed that these operators remain active (once the Logtalk pre-processor and runtime files are loaded) until the end of the Prolog session (this is the usual behavior of most Prolog compilers). Note that these operator definitions are compatible with the pre-defined operators in the Prolog ISO standard.

Sending a message to an object

Sending a message to an object is done by using the `::/2` infix operator:

```
| ?- Object::Message.
```

The message must match a public predicate declared for the receiving object or a Logtalk/Prolog built-in predicate, otherwise an error will be thrown (see the Reference Manual for details).

Broadcasting

In the Logtalk context, broadcasting is interpreted as the sending of the same message to a group of objects or the sending of several messages to the same object. Both needs can be achieved by using the message sending method described above. However, for convenience, Logtalk implements an extended syntax for message sending that makes programming easier in these situations.

If we wish to send several messages to the same object, we can write:

```
| ?- Object::(Message1, Message2, ...).
```

This is semantically equivalent to:

```
| ?- Object::Message1, Object::Message2, ... .
```

We can also write:

```
| ?- Object::(Message1; Message2; ...).
```

This will be semantically equivalent to writing:

```
| ?- Object::Message1; Object::Message2; ... .
```

To send the same message to a set of objects we can write:

```
| ?- (Object1, Object2, ...)::Message.
```

This will have the same semantics as:

```
| ?- Object1::Message, Object2::Message, ... .
```

If we want to use backtracking to try the same message over a set of objects we can write:

```
| ?- (Object1; Object2, ...)::Message.
```

This will be equivalent to:

```
| ?- Object1::Message; Object2::Message; ... .
```

Sending a message to *self*

While defining a predicate, we sometimes need to send a message to *self*, that is, to the same object that has received the original message. This is done in Logtalk through the `::/1` prefix operator:

```
::Message
```

We can also use the broadcasting constructs with this operator:

```
::(Message1, Message2, ...)
```

Or:

```
::(Message1; Message2; ...)
```

The message must match a public or protected predicate declared for the receiving object or a Logtalk/Prolog built-in predicate otherwise an error will be thrown (see the Reference Manual for details). If the message is sent from inside a category or if we are using private inheritance, then the message may also match a private predicate.

Calling an overridden predicate definition

When redefining a predicate, we sometimes have the need to call the inherited definition in the new code. This possibility, introduced by the Smalltalk language through the `super` primitive, is available in Logtalk through the `^^/1` prefix operator:

```
^^Predicate
```

Most of the time we will use this operator by instantiating the pattern:

```
Predicate :-
      ...,              % do something
      ^^Predicate,      % call inherited definition
      ... .             % do something more
```

Message sending and event generation

Every message sent using the `::/2` operator generates two events, one before and one after the message execution. Messages that are sent using the `::/1` (message to *self*) operator or the `^^/1` super mechanism described above do not generate any events. The rational behind this distinction is that messages to *self* and *super* calls are only used indirectly in the definition of methods or to execute additional messages with the same target object (represented by *self*). In other words, events are only generated when using an object's public interface. They can not be used to break object encapsulation.

If we need to generate events for a public message sent to *self*, then we just need to write something like:

```
Predicate :-
      ...,
      self(Self),        % get self reference
      Self::Message,     % send a message to self using ::/2
      ... .
```

If we also need the sender of the message to be other than the object containing the predicate definition, we can write:

```
Predicate :-
      ...,
      self(Self),        % get self reference
      {Self::Message},   % send a message to self using ::/2
      ... .              % sender will be the pseudo-object user
```

See the session on Event-driven programming for more details.

Objects

Logtalk objects should be regarded as a way to encapsulate and reuse predicate definitions. Instead of a single clause database containing all your code, Logtalk objects provide separated namespaces or databases allowing the partitioning of code in more manageable parts. Logtalk does not try to bring some sort of new dynamic state change concept to Logic Programming or Prolog.

In Logtalk, the only pre-defined object is the pseudo-object `user` that contains all the clauses in the Prolog database not contained in some Logtalk entity.

Objects, prototypes, classes and instances

Objects, prototypes, parents, classes, subclasses, superclasses, metaclasses, instances are all terms that always designate an object. Different names are used to emphasize the role on an object in a particular context. We use a term other than object when we want to make the relationship with other objects explicit. There are only three kinds of entities in Logtalk: objects, protocols and categories.

Defining a new object

We can define a new object in the same way we write Prolog code: by using a text editor. Each object (category or protocol) we define should be contained in its own text file. It is recommended that this text file be named after the object. By default, all Logtalk source files use the extension `.lgt` but this is optional and can be set in the configuration files. Compiled source files (by the Logtalk pre-processor) have, by default, a `.pl` extension. Again, this can be set to match the needs of a particular Prolog compiler in the corresponding configuration file. For instance, we may define an object named `vehicle` and save it in a `vehicle.lgt` source file that will be compiled to a `vehicle.pl` Prolog file.

Object names can be atoms or compound terms (if we are defining parametric objects, see below). Objects, categories and protocols share the same name space: we can not have an object with the same name as a protocol or a category.

Object code (directives and predicates) is textually enclosed between two Logtalk directives: `object/1-4` and `end_object/0`. The simplest object will be one that is self-contained, not depending on any other Logtalk entity:

```
:- object(Object).
      ...
:- end_object.
```

If an object implements one or more protocols then the opening directive will be:

```
:- object(Object,
      implements(Protocol)).
      ...
:- end_object.
```

An object can import one or more categories:

```
:- object(Object,
      imports(Category)).
      ...
:- end_object.
```

If an object implements protocols and imports categories then we will write:

```
:- object(Object,
      implements(Protocol),
      imports(Category)).
      ...
:- end_object.
```

In object-oriented programming objects are usually organized in hierarchies that enable interface and code sharing by inheritance. In Logtalk, we can construct prototype-based hierarchies by writing:

```
:- object(Prototype,
      extends(Parent)).
      ...
:- end_object.
```

We can also have class-based hierarchies by defining instantiation and specialization relations between objects. To define an object as a class instance we will write:

```
:- object(Object,
      instantiates(Class)).
      ...
:- end_object.
```

A class may specializes another class, its superclass:

```
:- object(Class,
      specializes(Superclass)).
      ...
:- end_object.
```

If we are defining a reflexive system where every class is also an object, we will probably be using the following pattern:

```
:- object(Class,
      instantiates(Metaclass),
      specializes(Superclass)).
      ...
:- end_object.
```

In short, an object can be a *stand-alone* object or be part of an object hierarchy. The hierarchy can be prototype-based (defined by extending other objects) or class-based (with instantiation and specialization relations). An object may also implement one or more protocols or import one or more categories.

A *stand-alone* object is always compiled as a prototype, that is, a self-describing object. If we want to use classes and instances, then we will need to specify at least an instantiation or a specialization relation. The best way to do this is to define a set of objects that provide the basis of a reflective system [Cointe 87, Moura 94]. For example:

```
:- object(object,               % default root of the inheritance graph
      instantiates(class)).     % contains predicates common to all objects
      ...
:- end_object.
```

```
:- object(class,                % default metaclass for all classes
      instantiates(class),      % contains predicates common to all instantiable classes
      specializes(abstract_class)).
      ...
:- end_object.

:- object(abstract_class,       % default metaclass for all abstract classes
      instantiates(class),      % contains predicates common to all classes
      specializes(object)).
      ...
:- end_object.
```

Note that with these instantiation and specialization relations `object`, `class` and `abstract_class` are, at the same time, classes and instances of some class. In addition, each object inherits its own predicates and the predicates of the other two objects without any inheritance loop problems.

We can have, in the same application, both prototype and class-based hierarchies (and freely exchange messages between all objects). We can not however mix the two types of hierarchies by, e.g., specializing an object that extends another object in this current Logtalk version.

Parametric objects

Parametric objects have a compound term for name instead of an atom. This compound term usually contains free variables that are instantiated when sending a message to the object. The object predicates can then be coded to depend on the parameters instantiation values. When an object state is set at object creation time and never changed, parameters provide a better solution than using the object's database via asserts. Parametric objects can also be used to attach a set of predicates to terms that share a common functor and arity.

In order to give access to an object parameters, Logtalk provides the `parameter/2` built-in local method:

```
:- object(Functor(Arg1, Arg2, ...)).

      ...

      Predicate :-
            ...,
            parameter(Number, Value),
            ... .
```

Note that we can't use this method with the message sending operators (`::/2`, `::/1` or `^^/1`). An alternative solution is to use the built-in local method `this/1`. For example:

```
:- object(foo(Arg)).

      ...

      bar :-
            ...,
            this(foo(Arg)),
            ... .
```

This is more efficient because the cost of the method `this/1` is only one single term unification, while `parameter/2` implies an `arg/3` call (an ISO Prolog Standard defined built-in predicate). The drawback is that we must check all calls of `this/1` if we change the object name.

Parametric objects are, by convention, stored in source files named after the objects with the object arity appended. For instance, if we define an object named `sort(Type)`, we may save it in a `sort1.lgt` text file. This way it is easy to avoid file name clashes when saving Logtalk entities that have the same functor but different arities.

Finding defined objects

We can find, by backtracking, all defined objects by calling the `current_object/1` built-in predicate with a non-instantiated variable:

```
| ?- current_object(Object).
```

This predicate can also be used to test if an object is defined by calling it with a valid object identifier (an atom or a compound term).

Creating a new object in runtime

An object can be dynamically created at runtime by using the `create_object/4` built-in predicate:

```
| ?- create_object(Object, Relations, Directives, Clauses).
```

The first argument, the name of the new object (a Prolog atom or compound term), should not match any existing entity name. The remaining three arguments correspond to the relations described in the opening object directive and to the object code contents (directives and clauses).

For instance, the call:

```
| ?- create_object(foo, [extends(bar)], [public(foo/1)], [foo(1), foo(2)]).
```

is equivalent to compiling and loading the object:

```
:- object(foo,
      extends(bar)).

      :- dynamic.

      :- public(foo/1).

      foo(1).
      foo(2).

:- end_object.
```

If we need to create a lot of (dynamic) objects at runtime, then is best to define a metaclass or a prototype with a predicate that will call this built-in predicate to make new objects. This predicate may provide automatic object name generation, name checking and accept object initialization options.

Abolishing an existing object

Dynamic objects can be abolished using the `abolish_object/1` built-in predicate:

```
| ?- abolish_object(Object).
```

The argument must be an identifier of a defined dynamic object, otherwise an error will be thrown.

Object directives

Object directives are used to set initialization goals and object properties and to document object dependencies on other Logtalk entities.

Object initialization

We can define a goal to be executed as soon as an object is (compiled and) loaded to memory with the `initialization/1` directive:

```
:- initialization(Goal).
```

The argument can be any valid Prolog or Logtalk goal, including a message to other object. For example:

```
:- object(foo).

      :- initialization(init).
      :- private(init/0).

      init :-
            ... .

:- end_object.
```

Or:

```
:- object(assembler).

      :- initialization(control::start).
      ...

:- end_object.
```

The initialization goal can also be a message to *self* in order to call an inherited or imported predicate. For example, assuming that we have a `monitor` category defining a `reset/0` predicate:

```
:- object(profiler,
      imports(monitor)).

      :- initialization(::reset).
      ...

:- end_object.
```

Note, however, that descendant objects do not inherit initialization directives. In this context, *self* denotes the object that contains the directive. Also note that by initialization we do not necessarily mean setting an object dynamic state.

Dynamic objects

As usually happens with Prolog code, an object can be either static or dynamic. An object created during the execution of a program is always dynamic. An object defined in a file can be either dynamic or static. Dynamic objects are declared by using the `dynamic/0` directive in the object source code:

```
:- dynamic.
```

Let us just remember that the loss of performance of the dynamic code is usually of considerable importance to the static code. We should only use dynamic objects when these need to be abolished during program execution. Also, note that we can declare and define dynamic predicates in a static object.

Object dependencies

Besides the relations declared in the object's opening directive, the predicate definitions contained in the object may imply other dependencies. These can be documented by using the `calls/1` and the `uses/1` directives.

The `calls/1` directive can be used when a predicate definition sends a message that is declared in a specific protocol:

```
:- calls(Protocol).
```

If a predicate definition sends a message to a specific object, this dependence can be declared with the `uses/1` directive:

```
:- uses(Object).
```

These two directives may be used by the Logtalk runtime to ensure that all needed entities are loaded when running an application.

Object documentation

An object can be documented with arbitrary user-defined information by using the `info/1` directive:

```
:- info(List).
```

See the Documenting Logtalk programs session for details.

Object relationships

Logtalk provides five sets of built-in predicates that enable us to query the system about the possible relationships that an object may have with other entities.

The built-in predicates `instantiates_class/2` and `instantiates_class/3` can be used to query all instantiation relations:

```
| ?- instantiates_class(Instance, Class).
```

Or, if we want to know the instantiation scope:

```
| ?- instantiates_class(Instance, Class, Scope).
```

Specialization relations can be found by using either the `specializes_class/2` or the `specializes_class/3` built-in predicates:

```
| ?- specializes_class(Class, Superclass).
```

Or, if we want to know the specialization scope:

```
| ?- specializes_class(Class, Superclass, Scope).
```

For prototypes, we can query extension relations with the `extends_object/2` or the `extends_object/3` built-in predicates:

```
| ?- extends_object(Object, Parent).
```

Or, if we want to know the extension scope:

```
| ?- extends_object(Object, Parent, Scope).
```

In order to find which objects import which categories we can use the built-in predicates `imports_category/2` or `imports_category/3`:

```
| ?- imports_category(Object, Category).
```

Or, if we want to know the importation scope:

```
| ?- imports_category(Object, Category, Scope).
```

To find which objects implement which protocols we can use the `implements_protocol/2` or the `implements_protocol/3` built-in predicates:

```
| ?- implements_protocol(Object, Protocol).
```

Or, if we want to know the implementation scope:

```
| ?- implements_protocol(Object, Protocol, Scope).
```

Note that, if we use a non-instantiated variable for the first argument, we will need to use the `current_object/1` built-in predicate to ensure that the entity returned is an object and not a category.

Object properties

We can find the properties of defined objects by calling the built-in predicate `object_property/2`:

```
| ?- object_property(Object, Property).
```

An object may have the property `static`, `dynamic`, or `built_in`. Dynamic objects can be abolished in runtime by calling the `abolish_object/1` built-in predicate.

The pseudo-object user

Logtalk defines a pseudo-object named `user` that contains all user predicate definitions not contained in a Logtalk entity. These predicates are implicitly declared public.

Protocols

Protocols enable the separation between interface and implementation: several objects can implement the same protocol and an object can implement several protocols. There are no pre-defined protocols in Logtalk.

Defining a new protocol

We can define a new protocol in the same way we write Prolog code: by using a text editor. Each protocol (object or category) we define should be contained in its own text file. It is recommended that this text file be named after the protocol. By default, all Logtalk source files use the extension `.lgt` but this is optional and can be set in the configuration files. Compiled source files (by the Logtalk pre-processor) have, by default, a `.pl` extension. Again, this can be set to match the needs of a particular Prolog compiler in the corresponding configuration file. For example, we may define a protocol named `listp` and save it in a `listp.lgt` source file that will be compiled to a `listp.pl` Prolog file.

Protocol names must be atoms. Objects, categories and protocols share the same name space: we can not have a protocol with the same name as an object or a category.

Protocol directives are textually enclosed between two Logtalk directives: `protocol/1-2` and `end_protocol/0`. The simplest protocol will be one that is self-contained, not depending on any other Logtalk entity:

```
:- protocol(Protocol).
      ...
:- end_protocol.
```

If a protocol extends one or more protocols, then the opening directive will be:

```
:- protocol(Protocol,
      extends(OtherProtocol)).
      ...
:- end_protocol.
```

Finding defined protocols

We can find, by backtracking, all defined protocols by using the `current_protocol/1` built-in predicate with a non-instantiated variable:

```
| ?- current_protocol(Protocol).
```

This predicate can also be used to test if a protocol is defined by calling it with a valid protocol identifier (an atom).

Creating a new protocol in runtime

We can create a new (dynamic) protocol in runtime by calling the Logtalk built-in predicate `create_protocol/3`:

```
| ?- create_protocol(Protocol, Relations, Directives).
```

The first argument, the name of the new protocol (a Prolog atom), should not match an existing entity name. The remaining two arguments correspond to the relations described in the opening protocol directive and to the protocol directives.

For instance, the call:

```
| ?- create_protocol(ppp, [extends(qqq)], [public(foo/1, bar/1)]).
```

is equivalent to compiling and loading the protocol:

```
:- protocol(ppp,
      extends(qqq)).

      :- dynamic.

      :- public(foo/1, bar/1).

:- end_protocol.
```

If we need to create a lot of (dynamic) protocols at runtime, then is best to define a metaclass or a prototype with a predicate that will call this built-in predicate in order to provide more sophisticated behavior.

Abolishing an existing protocol

Dynamic protocols can be abolished using the `abolish_protocol/1` built-in predicate:

```
| ?- abolish_protocol(Protocol).
```

The argument must be an identifier of a defined dynamic protocol, otherwise an error will be thrown.

Protocol directives

Protocol directives are used to set initialization goals and protocol properties.

Protocol initialization

We can define a goal to be executed as soon as a protocol is (compiled and) loaded to memory with the `initialization/1` directive:

```
:- initialization(Goal).
```

The argument can be any valid Prolog or Logtalk goal, including a message sending call.

Dynamic protocols

As usually happens with Prolog code, a protocol can be either static or dynamic. A protocol created during the execution of a program is always dynamic. A protocol defined in a file can be either dynamic or static. Dynamic protocols are declared by using the `dynamic/0` directive in the protocol source code:

```
:- dynamic.
```

Let us just remember that the loss of performance of the dynamic code is usually of considerable importance to the static code. We should only use dynamic protocols when these need to be abolished during program execution.

Protocol documentation

A protocol can be documented with arbitrary user-defined information by using the `info/1` directive:

```
:- info(List).
```

See the Documenting Logtalk programs session for details.

Protocol relationships

Logtalk provides two sets of built-in predicates that enable us to query the system about the possible relationships that a protocol has with other entities.

The built-in predicates `extends_protocol/2` and `extends_protocol/3` return all pairs of protocols so that the first one extends the second:

```
:- extends_protocol(Protocol1, Protocol2).
```

Or, if we want to know the extension scope:

```
:- extends_protocol(Protocol1, Protocol2, Scope).
```

To find which objects or categories implement which protocols we can call the `implements_protocol/2` or `implements_protocol/2` built-in predicates:

```
| ?- implements_protocol(ObjectOrCategory, Protocol).
```

Or, if we want to know the implementation scope:

```
| ?- implements_protocol(ObjectOrCategory, Protocol, Scope).
```

Note that, if we use a non-instantiated variable for the first argument, we will need to use the `current_object/1` or `current_category/1` built-in predicates to identify the kind of entity returned.

Protocol properties

We can find the properties of defined protocols by calling the `protocol_property/2` built-in predicate:

```
| ?- protocol_property(Protocol, Property).
```

A protocol may have the property `static`, `dynamic`, or `built_in`. Dynamic protocols can be abolished in runtime by calling the `abolish_protocol/1` built-in predicate.

Implementing protocols

Any number of objects or categories can implement a protocol. The syntax is very simple:

```
:- object(Object,
      implements(Protocol).
      ...
:- end_object.
```

Or, in the case of a category:

```
:- category(Object,
      implements(Protocol).
      ...
:- end_category.
```

To make all public predicates declared via an implemented protocol protected or to make all public and protected predicates private we prefix the protocol's name with the corresponding keyword. For instance:

```
:- object(Object,
      implements(private::Protocol).
      ...
:- end_object.
```

Or:

```
:- object(Object,
      implements(protected::Protocol).
      ...
:- end_object.
```

Omitting the scope keyword is equivalent to writing:

```
:- object(Object,
      implements(public::Protocol).
      ...
:- end_object.
```

The same rules apply to protocols implemented by categories.

Categories

Categories provide a way to encapsulate a set of related predicate definitions that do not represent an object and that only make sense when composed with other predicates. Categories may also be used to break a complex object in functional units. A category can be imported by several objects (without code duplication), including objects participating in prototype or class-based hierarchies. This concept of categories shares some ideas with Smalltalk-80 functional categories [Goldberg 83], Flavors mix-ins [Moon 86] (without implying multi-inheritance) and Objective-C categories [Cox 86]. There are no pre-defined categories in Logtalk.

Defining a new category

We can define a new category in the same way we write Prolog code: by using a text editor. Each category (object or protocol) we define should be contained in its own text file. It is recommended that this text file be named after the category. By default, all Logtalk source files use the extension `.lgt` but this is optional and can be set in the configuration files. Compiled source files (by the Logtalk pre-processor) have, by default, a `.pl` extension. Again, this can be set to match the needs of a particular Prolog compiler in the corresponding configuration file. For example, we may define a category named `documenting` and save it in a `documenting.lgt` source file that will be compiled to a `documenting.pl` Prolog file.

Category names must be atoms. Objects, categories and protocols share the same name space: we can not have a category with the same name as an object or a protocol.

Category code (directives and predicates) is textually enclosed between two Logtalk directives: `category/1-2` and `end_category/0`. The simplest category will be one that is self-contained, not depending on any other Logtalk entity:

```
:- category(Category).
      ...
:- end_category.
```

If a category implements one or more protocols then the opening directive will be:

```
:- category(Category,
      implements(Protocol)).
      ...
:- end_category.
```

Note that a category can't import other categories or inherit code from an object.

Finding defined categories

We can find, by backtracking, all defined categories by using the `current_category/1` Logtalk built-in predicate with a non-instantiated variable:

```
| ?- current_category(Category).
```

This predicate can also be used to test if a category is defined by calling it with a valid category identifier (an atom or a compound term).

Creating a new category in runtime

A category can be dynamically created at runtime by using the `create_category/4` built-in predicate:

```
| ?- create_category(Category, Directives, Clauses, Relations).
```

The first argument, the name of the new category – a Prolog atom – should not match with an existing entity name. The remaining three arguments correspond to the relations described in the opening category directive and to the category code contents (directives and clauses).

For instance, the call:

```
| ?- create_category(ccc, [implements(ppp)], [private(bar/1)], [(foo(X):-bar(X)), bar(1), bar(2)]).
```

is equivalent to compiling and loading the category:

```
:- category(ccc,
      implements(ppp)).

      :- dynamic.

      :- private(bar/1).

      foo(X) :-
            bar(X).

      bar(1).
      bar(2).

:- end_category.
```

If we need to create a lot of (dynamic) categories at runtime, then is best to define a metaclass or a prototype with a predicate that will call this built-in predicate in order to provide more sophisticated behavior.

Abolishing an existing category

Dynamic categories can be abolished using the `abolish_category/1` built-in predicate:

```
| ?- abolish_category(Category).
```

The argument must be an identifier of a defined dynamic category, otherwise an error will be thrown.

Category directives

Category directives are used to set initialization goals and category properties and to document category dependencies on other Logtalk entities.

Category initialization

We can define a goal to be executed as soon as a category is (compiled and) loaded to memory with the `initialization/1` directive:

```
:- initialization(Goal).
```

The argument can be any valid Prolog or Logtalk goal, including a message sending call.

Dynamic categories

As usually happens with Prolog code, a category can be either static or dynamic. A category created during the execution of a program is always dynamic. A category defined in a file can be either dynamic or static. Dynamic categories are declared by using the `dynamic/0` directive in the category source code:

```
:- dynamic.
```

Let us just remember that the loss of performance of the dynamic code is usually of considerable importance to the static code. We should only use dynamic categories when these need to be abolished during program execution.

Category dependencies

Besides the relations declared in the category's opening directive, the predicate definitions contained in the category may imply other dependencies. This can be documented by using the `calls/1` and the `uses/1` directives.

The `calls/1` directive can be used when a predicate definition sends a message that is declared in a specific protocol:

```
:- calls(Protocol).
```

If a predicate definition sends a message to a specific object, this dependence can be declared with the `uses/1` directive:

```
:- uses(Object).
```

These two directives can be used by the Logtalk runtime to ensure that all needed entities are loaded when running an application.

Category documentation

A category can be documented with arbitrary user-defined information by using the `info/1` directive:

```
:- info(List).
```

See the Documenting Logtalk programs session for details.

Category relationships

Logtalk provides two sets of built-in predicates that enable us to query the system about the possible relationships that a category can have with other entities.

The built-in predicates `implements_protocol/2` and `implements_protocol/3` find which categories implements which protocols:

```
| ?- implements_protocol(Category, Protocol).
```

Or, if we want to know the implementation scope:

```
| ?- implements_protocol(Category, Protocol, Scope).
```

Note that, if we use a non-instantiated variable for the first argument, we will need to use the `current_category/1` built-in predicate to ensure that the returned entity is a category and not an object.

To find which objects import which categories we can use the `imports_category/2` or `imports_category/3` built-in predicates:

```
| ?- imports_category(Object, Category).
```

Or, if we want to know the importation scope:

```
| ?- imports_category(Object, Category, Scope).
```

Note that several objects can import a category.

Category properties

We can find the properties of defined categories by calling the built-in predicate `category_property/2`:

```
| ?- category_property(Category, Property).
```

A category may have the property `static`, `dynamic`, or `built_in`. Dynamic categories can be abolished in runtime by calling the `abolish_category/1` built-in predicate.

Importing categories

Any number of objects can import a category. The syntax is very simple:

```
:- object(Object,
      imports(Category).
      ...
:- end_object.
```

To make all public predicates imported via a category protected or to make all public and protected predicates private we prefix the category's name with the corresponding keyword:

```
:- object(Object,
      imports(private::Category).
      ...
:- end_object.
```

Or:

```
:- object(Object,
      imports(protected::Category).
      ...
:- end_object.
```

Omitting the scope keyword is equivalent to writing:

```
:- object(Object,
      imports(public::Category).
      ...
:- end_object.
```

Predicates

Predicate declarations and definitions can be encapsulated inside objects and categories. Protocols can only contain predicate declarations.

Declaring predicates

All object (or category) predicates that we want to access from other objects must be explicitly declared. A predicate declaration must contain, at least, a scope directive. Other directives may be used to document the predicate or to ensure proper compilation of the predicate definitions.

Scope directives

A predicate can be public, protected or private. Public predicates can be called from any object. Protected predicates can only be called from the container object or from a container descendant. Private predicates can only be called from the container object.

The scope declarations are made using the `public/1`, `protected/1` and `private/1` directives. For example:

```
:- public(init/1).

:- protected(valid_init_option/1).

:- private(process_init_options/1).
```

Note that we do not need to write scope declarations for all defined predicates. If a predicate does not have a scope declaration, it is assumed that the predicate is private, although it will be invisible to the reflection methods and to the message and error handling mechanisms.

Mode directive

Many predicates cannot be called with arbitrary arguments with arbitrary instantiation status. The valid arguments and instantiation modes can be documented by using the `mode/2` directive. For instance:

```
:- mode(member(?term, +list), zero_or_more).
```

The first argument describes a valid calling mode. The minimum information will be the instantiation mode of each argument. There are four possible values (described in [ISO 95]):

`+`
: Argument must be instantiated.

`-`
: Argument must be a free (non-instantiated) variable.

`?`
: Argument can either be instantiated or free.

`@`
: Argument will not be modified.

Logtalk pre-processor declares these four mode atoms as prefix operators. This makes it possible to include type information for each argument like in the example above. Some of the possible type values are: `event`, `object`, `category`, `protocol`, `callable`, `term`, `nonvar`, `var`, `atomic`, `atom`, `number`, `integer`,

`float`, `compound`, and `list`. The first four are Logtalk specific. The remaining are common Prolog types. We can also use our own types that can be either atoms or compound terms.

The second argument documents the number of proofs (or solutions) for the specified mode. The possible values are:

`zero`
: Predicate always fails.

`one`
: Predicate always succeeds once.

`zero_or_one`
: Predicate either fails or succeeds.

`zero_or_more`
: Predicate has zero or more solutions.

`one_or_more`
: Predicate has one or more solutions.

`error`
: Predicate will throw an error (see below).

Mode declarations can also be used to document that some call modes will throw an error. For example, regarding the `arg/3` ISO Prolog built-in predicate, we may write:

```
:- mode(arg(-, -, +), error).
```

Note that most predicates have more than one valid mode implying several mode directives. For example, to document the possible use modes of the `atom_concat/3` ISO built-in predicate we would write:

```
:- mode(atom_concat(?atom, ?atom, +atom), one_or_more).
:- mode(atom_concat(+atom, +atom, -atom), zero_or_one).
```

Some old Prolog compilers supported some sort of mode directives to improve performance. To the best of my knowledge, there is no modern Prolog compiler supporting these kind of directive. The current version of the Logtalk pre-processor just parses and than discards this directive. Nevertheless, the use of mode directives is a good starting point to the documentation of your predicates.

Metapredicate directive

Some predicates may have arguments that will be called as goals. To ensure that these calls will be executed in the correct scope we need to use the `metapredicate/1` directive. For example:

```
:- metapredicate(findall(*, ::, *)).
```

The predicate arguments in this directive have the following meaning:

`::`
: Meta-argument that will be called as a goal.

`*`
: Normal argument.

This is similar to the declaration of metapredicates in the ISO draft for Prolog modules except that we use the atom `::` instead of `:` to be consistent with the message sending operators.

Since each Logtalk entity is independently compiled, this directive must be included in every object or category that contains a definition for the described predicate, even if the predicate declaration is inherited from another entity, to ensure proper compilation of meta-arguments.

Discontiguous directive

The clause of an object (or category) predicate may not be contiguous. In that case, we must declare the predicate discontiguous by using the `discontiguous/1` directive:

```
:- discontiguous(foo/1).
```

This is a directive that we should avoid using. It makes your code harder to read and it is not supported by some Prolog compilers.

Because each Logtalk entity is compiled independently from other entities, this directive must be included in every object or category that contains a definition for the described predicate (even if the predicate declaration is inherited from other entity).

Dynamic directive

An object (or category) predicate can be static or dynamic. By default, all object predicates are static. To declare a dynamic predicate we use the `dynamic/1` directive:

```
:- dynamic(foo/1).
```

Because each Logtalk entity is compiled independently from other entities, this directive must be included in every object or category that contains a definition for the described predicate (even if the predicate declaration is inherited from other entity). If we omit the dynamic declaration then the predicate definition will be compiled to static code. Note that any static object may declare and define dynamic predicates.

Documenting directive

A predicate can be documented with arbitrary user-defined information by using the `info/2` directive:

```
:- info(Functor/Arity, List).
```

The second argument is a list of `Key is Value` terms. See the Documenting Logtalk programs session for details.

Defining predicates

Object predicates

We define object predicates as we have always defined Prolog predicates, the only difference be that we have four more control structures (the three message sending operators plus the external call operator) to play with. For example, if we wish to define an object containing common utility list predicates like `append/2` or `member/2` we could write something like:

```
:- object(list).

      :- public(append/2).
      :- public(member/2).

      append([], L, L).
      append([H| T], L, [H| T2]) :-
            append(T, L, T2).
```

```
        member(H, [H| _]).
        member(H, [_| T]) :-
              member(H, T).

:- end_object.
```

Note that, abstracting from the opening and closing object directives and the scope directives, what we have written is plain Prolog. Calls in a predicate definition body default to the local predicates, unless we use the message sending operators or the external call operator. This enables easy conversion from Prolog code to Logtalk objects: we just need to add the necessary encapsulation and scope directives to the old code.

Category predicates

Because a category can be imported by several different objects, dynamic private predicates must be called using the `::/1` message sending operator. This ensures that the correct predicate definition will be used. For example, if we want to define a category implementing variables using destructive assignment we could write:

```
:- category(variable).

        :- public(get/2).
        :- public(set/2).

        :- private(value_/2).
        :- dynamic(value_/2).

        get(Var, Value) :-
              ::value_(Var, Value).

        set(Var, Value) :-
              ::retractall(value_(Var, _)),
              ::asserta(value_(Var, Value).

:- end_category.
```

This way, each importing object will have its own definition for the `value_/2` private predicate. Furthermore, the `get/2` and `set/2` predicates will always access/update the correct definition, contained in the object receiving the messages.

Built-in object predicates (methods)

Logtalk defines a set of built-in object predicates or methods to access message execution context, to find sets of solutions, to inspect objects and for database handling. Similar to Prolog built-in predicates, these built-in methods should not be redefined.

Local methods

Logtalk defines four built-in methods to access an object execution context. These methods are compiled in-line and can be freely used without worrying about performance penalties.

To find the object that received the message under execution we may use the `self/1` method. We may also retrieve the object that has sent the message under execution using the `sender/1` method.

The method `this/1` enables us to retrieve the name of the object that contains the code that is being executed instead of using the name directly. This helps to avoid breaking the code if we decide to change the object name and forget to change the name references.

Here is a short example including calls to these three object execution context methods:

```
:- object(test).

      :- public(test/0).

      test :-
            this(This),
            write('Executing  a predicate definition contained in '), writeq(This), nl,
            self(Self),
            write('to answer a message received by '), writeq(Self), nl,
            sender(Sender),
            write('that was sent by '), writeq(Sender), nl, nl.

:- end_object.


:- object(descendant,
      extends(test)).

:- end_object.
```

After compiling and loading these two objects, we can try the following goal:

```
| ?- descendant::test.

Executing  a predicate definition contained in test
to answer a message received by descendant
that was sent by user
yes
```

For parametric objects, the method `parameter/2` enables us to retrieve current parameter values (see the session on parametric objects for additional details). For example:

```
:- object(block(_Color)).

      :- public(test/0).

      test :-
            parameter(1, Color),
            write('Color parameter value  is '), writeq(Color), nl.

:- end_object.
```

After compiling and loading these two objects, we can try the following goal:

```
| ?- block(blue)::test.

Color parameter value is blue
yes
```

Database methods

Logtalk provides a set of built-in methods for object database handling similar to the usual database Prolog predicates: `abolish/1`, `asserta/1`, `assertz/1`, `clause/2`, `retract/1` and `retractall/1`.

Note that if we define clauses for a dynamic predicate inside a category, some of these methods will fail. Because several objects can import a category, category dynamic predicates can not be abolished and its clauses can not be access by `clause/2` or retracted. The clauses can however be overridden for an importing object by using the assert methods.

All solutions methods

The usual all solutions meta-predicates are pre-defined methods in Logtalk: `bagof/3`, `findall/3` and `setof/3`. There is also a `forall/2` method that implements generate and test loops.

Reflection methods

Logtalk provides two built-in methods for inspecting object predicates: `predicate_property/2`, that returns predicate properties, and `current_predicate/1`, that returns declared predicates. Together, they enable querying an object about predicate definitions. See below for a more detailed description of both methods.

Predicate properties

We can find the properties of visible predicates by calling the `predicate_property/2` built-in method. For example:

```
| ?- bar::predicate_property(foo(_), Property).
```

Note that this method respects the predicate's scope declarations. For instance, the above call will only return properties for public predicates.

An object's set of visible predicates is the union of all the predicates declared for the object with all the built-in methods and all the Logtalk and Prolog built-in predicates.

Possible predicate property values are:

- `public, protected, private`
- `static, dynamic`
- `built_in`
- `metapredicate(Mode)`
- `declared_in(Entity)`
- `defined_in(Entity)`

The last two properties, `declared_in/1` and `defined_in/1`, do not apply to built-in methods and Logtalk or Prolog built-in predicates.

Note that if a predicate is declared in a category imported by the object, it will be the category name – not the object name – which will be returned by the property `declared_in/1`.

Finding declared predicates

We can find, by backtracking, all visible user predicates by calling the `current_predicate/1` built-in method. This method respects the predicate's scope declarations. For instance, the following call:

```
| ?- some_object::current_predicate(Functor/Arity).
```

will only return user predicates that are declared public.

Inheritance

The inheritance mechanisms existent in object-oriented programming languages allow us the specialization of previously defined objects, avoiding the unnecessary repetition of code. In the context of logic programming, we can interpret inheritance as a form of theory extension: an object will virtually contain, besides its own predicates, all the predicates inherited from other objects.

Protocol inheritance

Protocol inheritance refers to the inheritance of predicate declarations (scope directives). These can be contained in objects, in protocols or in categories. Logtalk supports multi-inheritance of protocols: an object or a category may implement several protocols and a protocol may extend several protocols.

Search order for prototype hierarchies

The search order for predicate declarations is first the object, second the implemented protocols (and the protocols that these may extend), third the imported categories (and the protocols that they may implement), and last the objects that the object extends. This search is done in a depth-first way. If the object inherits two different declarations for the same predicate, only the first one will be considered.

Search order for class hierarchies

The search order for predicate declarations starts in the object classes. Following the classes declaration order, the search starts in the classes implemented protocols (and the protocols that these may extend), third the classes imported categories (and the protocols that they may implement), and last the superclasses of the object classes. This search is done in a depth-first way. If the object inherits two different declarations for the same predicate, only the first one will be considered.

Implementation inheritance

Implementation inheritance refers to the inheritance of predicate definitions. These can be contained in objects or in categories. Logtalk supports multi-inheritance of implementation: an object may import several categories or extend, specialize or instantiate several objects.

Search order for prototype hierarchies

The search order for predicate definitions is similar to the search for predicate declarations except that implemented protocols are ignored (they can only contain predicate directives).

Search order for class hierarchies

The search order for predicate definitions is similar to the search for predicate declarations except that implemented protocols are ignored (they can only contain predicate directives).

Inheritance versus predicate redefinition

When we define a predicate that is already inherited from other object, the inherited definitions are hidden by the new definitions. This is called overriding inheritance: a local definition overrides any inherited ones. For example, assume that we have the following two objects:

```
:- object(root).

      :- public(bar/1).
      :- public(foo/1).

      bar(root).

      foo(root).

:- end_object.


:- object(descendant,
      extends(root)).

      foo(descendant).

:- end_object.
```

After compiling and loading these objects, we can check the overriding behavior by trying the following queries:

```
| ?- root::(bar(Bar), foo(Foo)).

Bar = root
Foo = root
yes


| ?- descendant::( bar(Bar ), foo(Foo)).

Bar = root
Foo = descendant
yes
```

However, we can explicitly program other behaviors. Let us see a few examples.

Specialization inheritance

Specialization of inherited definitions: the new definition uses the inherited definitions, adding to this new code. This is done by calling the `^^/1` operator in the new definition.

```
:- object(root).

      :- public(init/0).

      init :-
            write('root init'), nl.

:- end_object.
```

```
:- object(descendant,
      extends(root)).

      init :-
            write('descendant init'), nl,
            ^^init.

:- end_object.


| ?- descendant::init.

root init
descendant init
yes
```

Union inheritance

Union of the new with the inherited definitions: all the definitions are taken into account, the calling order being defined by the inheritance mechanisms. This can be accomplished by writing a clause that just calls, using the `^^/1` operator, the inherited definitions. The relative position of this clause among the other definition clauses sets the calling order for the local and inherited definitions.

```
:- object(root).

      :- public(foo/1).

      foo(1).
      foo(2).

:- end_object.


:- object(descendant,
      extends(root)).

      foo(3).
      foo(Foo) :-
            ^^foo(Foo).

:- end_object.


| ?- descendant::foo(Foo).

Foo = 3 ;
Foo = 1 ;
Foo = 2 ;
no
```

Selective inheritance

Hiding of some of the inherited definitions or differential inheritance: this form of inheritance is normally used in the representation of exceptions to generic definitions. Here we will need to use the `^^/1` operator to test and possibly reject some of the inherited definitions.

```
:- object(bird).

      :- public(mode/1).

      mode(walks).
      mode(flies).

:- end_object.
```

```
:- object(penguin,
      extends(bird)).

      mode(swims).
      mode(Mode) :-
            ^^mode(Mode),
            Mode \= flies.

:- end_object.

| ?- penguin::mode(Mode).

Mode = swims ;
Mode = walks ;
no
```

Public, protected and private inheritance

To make all public predicates declared via implemented protocols, imported categories, or inherited objects protected or to make all public and protected predicates private we prefix the entity's name with the corresponding keyword. For instance:

```
:- object(Object,
      implements(private::Protocol).
      % all the public and protected predicates in
      % Protocol become Object's private predicates
      ...
:- end_object.
```

Or:

```
:- object(Class,
      specializes(protected::Superclass).
      % all the Superclass public predicates
      % become Object's protected predicates
      ...
:- end_object.
```

Omitting the scope keyword is equivalent to using the public scope keyword. For example:

```
:- object(Object,
      imports(public::Category).
      ...
:- end_object.
```

This is the same as:

```
:- object(Object,
      imports(Category).
      ...
:- end_object.
```

This way we ensure backward compatibility with older Logtalk versions and a simplified syntax when protected or private inheritance is not used.

Composition versus multiple inheritance

It is not possible to discuss inheritance mechanisms without referring to the long and probably endless debate on single versus multiple inheritance. The single inheritance mechanism can be implemented in a very

efficient way, but it imposes several limitations on reusing, even if the multiple characteristics we intend to inherit are orthogonal. On the other hand, the multiple inheritance mechanisms are attractive in their apparent capability of modeling complex situations. However, they include a potential for conflict between inherited definitions whose variety does not allow a single and satisfactory solution for all the cases.

Until now, no solution that we might consider satisfactory for all the problems presented by the multiple inheritance mechanisms has been found. From the simplicity of some extensions that use the Prolog search strategy like [McCabe 92] or [Moss 94] and to the sophisticated algorithms of CLOS [Bobrow 88], there is no adequate solution for all the situations. Besides, the use of multiple inheritance carries some complex problems in the domain of software engineering, particularly in the reuse and maintenance of the applications. All these problems are substantially reduced if we preferably use in our software development composition mechanisms instead of specialization mechanisms [Taenzer 89]. Multiple inheritance can and should be seen more as a useful analysis and project abstraction, than as an implementation technique [Shan 93].

Nevertheless, Logtalk supports multi-inheritance by enabling an object to extend, instantiate, or specialize more than one object. Beware however that the current Logtalk release does not provide any mechanism for selecting a specific inheritance path or for dealing with inheritance conflicts. The multi-inheritance support implementation does not compromise performance when we use single-inheritance.

Event-driven programming

The addition of event-driven programming capacities to the Logtalk system is based on a simple but powerful idea [Moura 94]:

> The computations must result, not only from message sending, but also from the **observation** of message sending.

The need to associate computations to the occurrence of events was very early recognized in several knowledge representation languages, in some programming languages [Stefik 86, Moon 86], and in the implementation of operative systems [Tanenbaum 87] and graphical user interfaces.

With the integration between object-oriented and event-driven programming, we intend to achieve the following goals:

· Minimize the coupling between objects. An object should only contain what is intrinsic to it. If an object observes another object, that means that it should depend only on the (public) protocol of the object observed, and not on the implementation of that same protocol.

· Provide a framework for building reflexive systems in Logtalk based on the dynamic behavior of objects in complement to the reflective information of the object's contents and relations.

Definitions

The words *event* and *monitor* have multiple meanings in computer science, so, to avoid misunderstandings, it is advisable that we start by defining them in the Logtalk context.

Event

In an object-oriented system, all computations start through message sending. It thus becomes quite natural to declare that the only event that can occur in this kind of system is precisely the sending of a message. An event can thus be represented by the ordered tuple `(Object, Message, Sender)`.

If we consider message processing an indivisible activity, we can interpret the sending of a message and the return of the control to the object that has sent the message as two distinct events. This distinction allows us to have a more precise control over a system dynamics. In Logtalk, these two types of events have been named `before` and `after`, respectively for message sending and returning. Therefore, we end up by representing an event by the ordered tuple `(Event, Object, Message, Sender)`.

The implementation of the event notion in Logtalk enjoys the following properties:

Independence between the two types of events. We can choose to watch only one event type or to process each one of the events associated to a message sending in an independent way.

All events are automatically generated by the message sending mechanism. The task of generating events is accomplished, in a transparent way, by the message sending mechanism. The user just defines which are the events in which he is interested.

The events watched at any moment can be dynamically changed during program execution. The notion of event allows the user not only to have the possibility of observing, but also of controlling and modifying

an application behavior, namely by dynamically changing the observed events during program execution. It is our goal to provide the user with the possibility of modeling the largest possible number of situations.

Monitor

Complementary to the notion of event is the notion of monitor. A monitor is an object that is automatically notified by the message sending mechanisms whenever certain events occur. A monitor should naturally define the actions to be carried out whenever a monitored event occurs.

The implementation of the monitor notion in Logtalk enjoys the following properties:

Any object can act as a monitor. The monitor status is a role that any object can perform during its existence. The minimum protocol necessary is declared in protocol `event_handlersp`. An extended protocol is available in protocol `monitorp`.

Unlimited number of monitors for each event. Several monitors can observe the same event because of distinct reasons. Therefore, the number of monitors per event is bounded only by the available computing resources.

The monitor status of an object can be dynamically changed in runtime. This property does not imply that an object must be dynamic to act as a monitor (the monitor status of an object is not stored in the object).

The execution of actions, defined in a monitor, associated to each event, never affects the term that denotes the message involved. In other words, if the message contains non-instantiated variables, these are not affected by the acting of monitors associated to the event.

Event generation

For each message that is sent (using the `::/2` message sending mechanism) the runtime system automatically generates two events. The first — `before event` — is generated when the message is sent. The second — `after event` — is generated after the message has successfully been executed.

Communicating events to monitors

Whenever a spied event occurs, the message sending mechanisms call the corresponding event handlers directly for all registered monitors. These calls are made bypassing the message sending primitives in order to avoid potential endless loops. The event handlers consist in user definitions for pre-declared public predicates (one for each event kind; see below for more details).

Performance concerns

The existence of monitored messages should not affect the processing of the remaining messages. On the other hand, for each message that has been sent, the system must verify if its respective event is monitored. This verification clearly must be ideally performed in constant time and independently from the number of monitored events. The event representation takes advantage of the first argument indexing performed by most Prolog compilers, which ensure - in the general case - an access in constant time.

Monitor semantics

The established semantics for monitor actions consists on considering its success as a necessary condition so that a message can succeed:

- All actions associated to events of type `before` must succeed, so that the message processing can start.
- All actions associated to events of type `after` also have to succeed so that the message itself succeeds. The failure of any action associated to an event of type `after` forces backtracking over the message execution (the failure of a monitor never causes backtracking over the preceding monitor actions).

Note that this is the most general choice. If we wish a transparent presence of monitors in a message processing, we just have to define the monitor actions in such a way that they never fail (which is very simple to accomplish).

Activation order of monitors

Ideally, whenever there are several monitors defined for the same event, the calling order should not interfere with the result. However, this is not always possible. In the case of an event of type `before`, the failure of a monitor prevents a message from being sent and prevents the execution of the remaining monitors. In case of an event of type `after`, a monitor failure will force backtracking over message execution. Different orders of monitor activation can therefore lead to different results if the monitor actions imply object modifications unrecoverable in case of backtracking. Therefore, the order for monitor activation must be always taken as arbitrary. In effect, to suppose or to try to impose a specific sequence implies a global knowledge of an application dynamics, which is not always possible. Furthermore, that knowledge can reveal itself as incorrect if there is any changing in the execution conditions. Note that, given the independence between monitors, it does not make sense that a failure forces backtracking over the actions previously executed.

Event handling

Logtalk provides three built-in predicates for event handling. These predicates enable you to find what events are defined, to define new events and to abolish events when they are no longer needed. If you plan to use events extensively in your application, then you should probably define a set of objects that use the built-in predicates described below to implement more sophisticated and high-level behavior.

Finding defined events

The events that are currently defined can be retrieved using the Logtalk built-in predicate `current_event/5`:

```
| ?- current_event(Event, Object, Message, Sender, Monitor).
```

Note that this predicate will return a **set** of matching events if some of the returned arguments are free variables or contain free variables.

Defining new events

New events can be defined using the Logtalk built-in predicate `define_events/5`:

```
| ?- define_events(Event, Object, Message, Sender, Monitor).
```

Note that if any of the arguments is a free variable or contains free variables, this call will define the **set** of matching events.

Abolishing defined events

Events that are no longer needed may be abolished using the `abolish_events/5` built-in predicate:

```
| ?- abolish_events(Event, Object, Message, Sender, Monitor).
```

If called with free variables, this goal will remove all matching events.

Defining event handlers

There are two pre-declared public predicates, `before/3` and `after/3`, that are automatically called to handle `before` and `after` events. Any object that plays the role of monitor should define one or both of these event handler methods:

```
before(Object, Message, Sender) :-
      ... .

after(Object, Message, Sender) :-
      ... .
```

The arguments in both methods are instantiated by the message sending mechanisms when a spied event occurs. For example, assume that we want to define a monitor called `tracer` that will track any message sent to an object by printing a describing text to the standard output. Its definition could be something like:

```
:- object(tracer).

      before(Object, Message, Sender) :-
            write('call: '), writeq(Object), write(' <-- '), writeq(Message),
            write(' from '), writeq(Sender), nl.

      after(Object, Message, Sender) :-
            write('exit: '), writeq(Object), write(' <-- '), writeq(Message),
            write(' from '), writeq(Sender), nl.

:- end_object.
```

Assume that we also have the following object:

```
:- object(any).

      :- public(bar/1) .
      :- public(foo/1) .

      bar(bar).

      foo(foo).

:- end_object.
```

After compiling and loading both objects, we can start tracing every message sent to any object by calling the `define_events/5` built-in predicate:

```
| ?- define_events(_, _, _, _, tracer).

yes
```

From now on, every message sent to any object will be traced to the standard output stream:

```
| ?- any::bar(X).

call: any <-- bar(X) from user
exit: any <-- bar(bar) from user
X = bar

yes
```

To stop tracing, we can use the `abolish_events/5` built-in predicate:

```
| ?- abolish_events(_, _, _, _, tracer).

yes
```

Error handling

All error/exception handling is done in Logtalk by using the ISO defined `catch/3` and `throw/1` predicates [ISO 95]. Some Prolog compilers do not implement these predicates or, if they do, the implementation is not compatible with the standard. Furthermore, the nature of these predicates does not allow their definition by the user. For these reasons, we should check our Prolog compiler before trying to add error-handling code to your Logtalk applications.

Errors thrown by Logtalk defined built-in predicates have the following format:

```
error(Error, Call)
```

For example:

```
error(type_error(object_identifier, 33), current_object(33))
```

Errors thrown while processing a message have the following format:

```
error(Error, Message, Sender)
```

For example:

```
error(permission_error(modify, private_predicate, bar(_)), foo::abolish(bar/1), user)
```

Compiler warnings and errors

The Logtalk pre-processor/compiler uses the `read_term/3` ISO Prolog defined built-in predicate to read and process a Logtalk source file. One consequence of this is that invalid Prolog terms or syntax errors may abort the compilation process with limited information given to the user (due to the inherent limitations of the `read_term/3` predicate).

If all the (Prolog) terms in a source file are valid, then there is a set of errors or potential errors, described below, that the pre-processor will try to detect and report.

Singleton variables

Singleton variables in a clause are often misspell variables and, as such, one of the most common errors when programming in Prolog. If your Prolog compiler complies with the Prolog ISO standard, or at least supports the ISO predicate `read_term/3` called with the option `singletons(S)`, then the Logtalk pre-processor/compiler will warn us of any singleton it finds while compiling a Logtalk entity.

Redefinition of Prolog built-in predicates

The Logtalk pre-processor/compiler will warn us of any redefinition of a Prolog built-in predicate inside an object or category. Sometimes the redefinition is intended. In other cases, the user may not be aware that the subjacent Prolog compiler may already provide the predicate as a built-in or we may want to ensure code portability among several Prolog compilers with different sets of built-in predicates.

Redefinition of Logtalk built-in predicates

Similar to the redefinition of Prolog built-in predicates, the Logtalk compiler will warn us if we try to redefine a Logtalk built-in. The redefinition will probably be an error in almost all (if not all) cases.

Redefinition of Logtalk built-in methods

An error will be thrown if we attempt to redefine a Logtalk built-in method inside an entity. The default behavior is to report the error and abort the compilation of the offending entity.

Misspell calls of local predicates

A warning will be reported if Logtalk finds (in the body of a predicate definition) a call to a local predicate that is not defined, built-in (either in Prolog or in Logtalk) or declared dynamic. In most cases these calls are simple misspell errors.

Other warnings and errors

The Logtalk pre-processor/compiler will throw an error if it finds a predicate clause or a directive that cannot be parsed. The default behavior is to report the error and abort the compilation of the offending entity.

Runtime errors

This session briefly describes runtime errors that result from misuse of Logtalk built-in predicates, built-in methods or from message sending. For a complete and detailed description of runtime errors please consult the Reference Manual.

Logtalk built-in predicates

All Logtalk built-in predicates check the type and mode of the calling arguments, throwing an exception in case of misuse.

Logtalk built-in methods

Every Logtalk built-in method checks the type and mode of the calling arguments, throwing an exception in case of misuse.

Message sending

The message sending mechanisms always check if the receiver of a message is a defined object and if the message corresponds to a declared predicate within the scope of the sender.

Documenting Logtalk programs

Logtalk automatically generates a documentation file for each compiled entity (object, protocol, or category) in XML format. Contents of the XML file include the entity name, type, and compilation mode (static or dynamic), the entity relations with other entities, and a description of any declared predicates (name, compilation mode, scope, etc).

The XML documentation files can be enriched with arbitrary user-defined information, either about an entity or about its predicates, by using the two directives described below.

Documenting directives

Logtalk supports two documentation directives for providing arbitrary user-defined information about an entity or a predicate. These two directives complement other Logtalk directives that also provide important documentation information like `uses/1`, `calls/1`, or `mode/2`.

Entity directives

Arbitrary user-defined entity information can be represented using the `info/1` directive:

```
:- info([
      Key1 is Value1,
      Key2 is Value2,
      ...]).
```

In this pattern, keys should be atoms and values should be ground terms. The following keys are pre-defined and may be processed specially by Logtalk:

`comment`
: Comment describing entity purpose (an atom).

`authors`
: Entity authors (an atom).

`version`
: Version number (a number).

`date`
: Date of last modification (formatted as Year/Month/Day).

`parnames`
: Parameter names for parametric entities (a list of atoms).

For example:

```
:- info([
      version is 2.1,
      authors is 'Paulo Moura',
      date is 2000/4/20,
      comment is 'Building representation.',
      diagram is 'UML Class Diagram #312']).
```

Use only the keywords that make sense for your application and remember that you are free to invent your own keywords.

Predicate directives

Arbitrary user-defined predicate information can be represented using the `info/2` directive:

```
:- info(Functor/Arity, [
      Key1 is Value1,
      Key2 is Value2,
      ...]).
```

Keys should be atoms and values should be ground terms. The following keys are pre-defined and may be processed specially by Logtalk:

`comment`
: Comment describing predicate purpose (an atom).

`argnames`
: Names of predicate arguments for pretty print output (a list of atoms).

`allocation`
: Objects where we should define the predicate. Some possible values are `container`, `descendants`, `instances`, `classes`, `subclasses`, and `any`.

`redefinition`
: Describes if the predicate can be redefined and in what way. Some possible values are `never`, `free`, `specialize`, `call_super_first`, `call_super_last`.

For example:

```
:- info(color/1, [
      comment is 'Table of defined colors.',
      argnames is ['Color'],
      constraint is 'Only a maximum of four visible colors allowed.']).
```

Use only the keywords that make sense for your application and remember that you are free to invent your own keywords.

Processing and viewing documenting files

The XML documenting files are automatically generated when you compile a Logtalk entity. For example, assuming the default filename extensions, compiling a `sort1.lgt` file generates a `sort1.pl` Prolog file and a `sort1.xml` XML file. The filename extension for each kind of file can be changed in the configuration files via the `lgt_file_extension/2` predicate.

Each XML file contains references to two other files: `logtalk.dtd`, a DTD file describing the XML file structure, and a XSL style sheet file responsible for converting the XML files to some desired format like HTML. The name of the XSL file can be changed in the configuration files via the `lgt_file_name/2` predicate. The default value is `logtalk.xsl`, a XSL file that converts Logtalk XML files to HTML files. The HTML output refers a CSS file, `logtalk.css`, which specifies how the HTML code will be rendered. The three default `logtalk.*` files are contained in the `xml` sub-directory in the Logtalk installation directory. This directory may also contain other files for specific XSLT tools or for converting XML to other formats besides HTML. Please read the `NOTES` file included in the directory for details.

By default, all file references use relative paths, assuming that the `.xml` documentation files and the `.dtd`, `.xsl`, and `.css` files reside in the same directory. This can accomplished either by copying the DTD and style files to the your application compiling directory or by using file aliases or symbolic links, depending on the operating system that you are using. Of course, you may also copy the `*.xml` files to the `xml` sub-directory.

To view the XML documenting files you can open them in a web browser that supports the XML, XSL, CSS 1, and HTML 4 standards or use a XSLT tool to compile the `.xml` files to `.html` files. You can also write new XSL files to convert the XML code to alternative formats like LaTeX or any other desired format.

Logtalk configuration

The Logtalk system is entirely written in Prolog, without resorting to other programming languages. In spite of that, the fact that the ISO standard [ISO 95] has only recently been approved causes a lot of problems when we try to ensure compatibility with most of the existing Prolog compilers [Moura 99]. If most syntax problems can be easily solved, the same does not happen when we talk about built-in predicates, where we face several difficulties. In the first place, many compilers do not implement the total of the predicates defined in ISO standard, which can be considered quite minimalist in this respect. Sometimes they implement those predicates in a way that does not match the standard or other compilers, thus concurring to the existence of subtle errors quite difficult to detect. To make things worse, it is not possible to replace some of the absent predicates and directives with our own definitions, due to the very nature of those predicates and directives. Consequently, some of the Logtalk system features (e.g., error handling) may not work in some compilers.

The adopted solution consists on developing a universal Logtalk version, linked to a given compiler through the definition of a small set of predicates and operators, assembled in a configuration file. These predicates can be divided in two groups. The first group contains Logtalk specific predicates. The second group tries to implement predicates that, although they are defined in the approved ISO Prolog Standard, are not available in the specific compiler we want to use. This manual session explains how to build a configuration file for a Prolog compiler.

Hardware & software requirements

Computer and operating system

Logtalk is compatible with almost any machine/operating system with a modern Prolog compiler available. Currently, my main development environment is an upgraded Apple PowerMacintosh 7600 running Mac OS 9 and LinuxPPC; most used compilers are the Mac port of GNU Prolog and, under Linux, YAP and SWI Prolog. Being written in Prolog and distributed in source form, the only issue regarding operating system compatibility are the end-of-line codes in the source text files! Most example source file names do not fit in the 8+3 MS-DOS file length limitation, so this may prevent those examples from running under this operating system (or in any other system with similar limitations).

Prolog compiler

In writing Logtalk I have tried to follow the recently approved Prolog ISO standard whenever possible. Capabilities needed by Logtalk that are not defined in the ISO standard are:

- access to predicate properties (`dynamic`, `static`, `built_in`)
- `abort/0` predicate

Logtalk needs access to the predicate property `built_in` to properly compile objects and categories that contain Prolog built-in predicates calls. In addition, some Logtalk built-ins need to know the dynamic/static status of predicates to ensure correct application. The current draft for the ISO standard for Prolog modules defines a `predicate_property/2` predicate that is already implemented by most Prolog compilers. The Logtalk compiler and runtime use the `abort/0` predicate for error handling. Note that if these capabilities are not built-in the user cannot easily define them.

For optimal performance, Logtalk requires that the Prolog compiler support **first-argument indexing** for both static and dynamic code.

Because most Prolog implementers are slowly moving toward more ISO compliant compilers, it is advisable that you try to use the most recent version of your favorite Prolog compiler.

Configuration files

Configuration files provide the glue code between the Logtalk pre-processor/runtime and a Prolog compiler. Each configuration file contains two sets of predicates: ISO Prolog standard predicates and directives not built-in in the target Prolog compiler and Logtalk-specific predicates.

Logtalk already includes ready to use configuration files for most Prolog compilers. However, you may need to write your own configuration file if one is not available for your Prolog compiler. In most cases, you can borrow code from some of the predefined configuration files. If you send me your configuration file, with a reference to the target Prolog compiler, maybe I can include it in the next release of Logtalk.

Start by making a copy of the file `configs/template.config`. Carefully check (or complete if needed) each listed definition. If your Prolog compiler conforms to the ISO standard, this task should only take you a few minutes.

If you are unsure that your Prolog compiler provides all the ISO predicates needed by Logtalk, try to run the system by setting the unknown predicate error handler to report as an error any call to a missing predicate. Better yet, switch to a modern, ISO compliant, Prolog compiler.

Installing and running Logtalk

Installing Logtalk

The Logtalk system can be installed in any directory that is accessible to the user. The installation process consists merely in decompressing a file that will lead to a new directory with the structure/contents described below. The decompression process naturally depends on the operative system that you are using.

Mac OS

The Macintosh version is included in the file `lgt2xx.sea.bin`, a `MacBinary` encoded, self-extracting archive. Your web browser should automatically decode the file, giving you a `.sea` self-extracting archive that you double-click to install Logtalk. If not, drag and drop the `.bin` file in a utility like `StuffIt Expander` or `MacBinaryII+`.

Linux, Unix

The Linux/Unix version is included in the file `lgt2xx.tar.gz`. In order to decompress and install the system we can use the following commands:

```
% gunzip lgt2xx.tar.gz
% tar -xvf lgt2xx.tar
```

This will create a sub-directory named `lgt2xx` in your current directory.

OS/2, Windows 95/NT

The OS/2 and Windows 95/NT version is included in the file `lgt2xx.zip`. The file can be decompressed using a utility like `unzip`:

```
unzip lgt2xx.zip
```

Other operating systems

Almost all files in the Logtalk distribution are text files. The only difference between the source files, other than the compressing formats, is the end-of-line codes: Macintosh uses a carriage return, Unix uses a line feed, OS/2, Windows 95/NT uses both a carriage return and a line feed. This should make it easier to install Logtalk under other operating systems.

Directories and files organization

In the Logtalk installation directory, you will find the following files and directories:

`LICENSE` - Logtalk user license
`QUICK_START` - Quick start instructions for those that do not like to read manuals
`README` - several useful information

`RELEASE_NOTES` - release notes for this version
`UPGRADING` - instructions about how to upgrade your programs to the current Logtalk version
`compiler`
    `NOTES` - notes on the current status of the compiler
    `...` - compiler source files
`configs`
    `NOTES` - notes on the provided configuration files
    `template.config` - template configuration file
    `...` - specific configuration files
`examples`
    `NOTES` - short description of the provided examples
    `bricks`
        `NOTES` - example description and other notes
        `SCRIPT` - step by step example tutorial
        `bricks.loader` - loader utility file for the example objects
        `...` - bricks example source files
    `...` - other examples
`manuals`
    `NOTES` - notes on the provided documentation
    `bibliography.html` - bibliography
    `glossary.html` - glossary
    `index.html` - root document for all documentation
    `...` - other documentation files
`xml`
    `NOTES` - notes on the automatic generation of XML documentation files
    `logtalk.css` - CSS style sheet file for the HTML output of the XSLT conversion of the XML files
    `logtalk.dtd` - DTD file describing the structure of the XML files
    `logtalk.xsl` – XSLT transformation style sheet to output HTML code from the XML files
    `...` - other XML related files

Configuration files

Logtalk includes several configuration files for most academic and commercial Prolog compilers. If a configuration file is not available for the compiler that you intend to use, then you need to build a new one, starting from the included `template.config` file.

Since most Prolog compilers are moving closer to the ISO Prolog standard [ISO 95], it is advisable that you try to use a recent version of your Prolog compiler of choice.

Logtalk compiler and runtime

The `compiler` sub-directory contains the Prolog source file(s) that implement the Logtalk pre-processor/compiler and the Logtalk runtime. The compiler and the runtime may be split in two (or more) separate files or combined in a single file, depending on the Logtalk release that you are installing.

Examples

Logtalk 2.x contains new implementations of some of the examples provided with previous 1.x versions. The sources of each one of these examples can be found included in a subdirectory with the same name, inside the directory `examples`. The majority of these examples include a file named `SCRIPT` that contains cases of simple utilization. Some examples may depend on other examples to work properly. Read the corresponding `NOTES` file for details before running an example.

Logtalk source files

Each Logtalk entity (object, category or protocol) is contained in a text file named after the entity. The extension `.lgt` is normally used. The Logtalk pre-processor compiles these files to plain Prolog, replacing the `.lgt` extension with `.pl` (the default Prolog extension). If your Prolog compiler expects the Prolog source filenames to end with a specific, different extension, you can set it in the corresponding configuration file.

Loader utility files

Most example directories contain a Prolog utility file that can be used to load all included source files. These loader utility files are named `<dir name>.loader` and should be consulted like any ordinary Prolog file.

Usually these files contain a call to the Logtalk built-in predicate `logtalk_load/1`, wrapped inside an `initialization/1` directive (to ensure ISO compatibility). For instance, if your code is split in three Logtalk source files named `source1.lgt`, `source2.lgt`, and `source3.lgt`, then the contents of your loader file will be:

```
:- initialization(
      logtalk_load([
      source1,
      source2,
      source3])).
```

These loader files may not work without modifications depending on the way your Prolog compiler deals with folders/directories. Most of the time you will need to **set the working directory** to be the one that contains the loader file in order to get the example source files loaded. Unfortunately, there is no portable way for us to do that from inside Logtalk due to differences between operating systems and lack of adequate operating system access support in some Prolog compilers.

Running a Logtalk session

We run Logtalk inside a normal Prolog session, after loading the needed files. Logtalk extends but does not modify your Prolog compiler. We can freely mix Prolog queries with the sending of messages and our programs can be made of both normal Prolog clauses and object definitions.

Starting Logtalk

To start a Logtalk session just:

1. Start Prolog.
2. Load the appropriate configuration file for your compiler. Configuration files for most common Prolog compilers can be found in the `configs` subdirectory.
3. Load the Logtalk compiler/pre-processor and runtime files contained in the `compiler` subdirectory.

Note that the both configuration files and compiler/pre-processor files are Prolog files. The predicate called to load them depends on your Prolog compiler. In case of doubt, look at the definition of the predicate `lgt_load_prolog_code/1` in the configuration file.

Compiling and loading your programs

Your programs will be made of source files containing your objects, protocols and categories. After changing the working directory to the one containing your files, you can compile a source file by calling the Logtalk built-in predicate `logtalk_compile/1`:

```
| ?- logtalk_compile(my_source_file).
```

This predicate runs the pre-processor on the argument file and, if no fatal errors are found, outputs a Prolog source file that can then be consulted or compiled in the usual way by your Prolog compiler.

To compile and also load to memory a source file we can use the Logtalk built-in predicate `logtalk_load/1`:

```
| ?- logtalk_load(my_source_file).
```

This predicate works in the same way of the predicate `logtalk_compile/1` but also loads the compiled file to memory (the Prolog predicate called for this is defined in the configuration file).

Both predicates expect an atom, usually the entity name, as an argument. The Logtalk source file name extension, as defined in the configuration file, should be omitted.

If you have more than a few source files then you may want to use a loader utility file containing the calls to the `logtalk_load/1` predicate (see the description above). Consulting or compiling the loader file will then compile and load all your Logtalk entities into memory.

Programming in Logtalk

Logtalk scope

Logtalk, as an object-oriented extension to Prolog, shares with it the same preferred areas of application but also extends them with those areas where object-oriented features provide an advantage compared to plain Prolog. Among these areas we have:

Object-oriented programming teaching and researching: Logtalk smooth learning curve, combined with support for prototype and class-based programming, protocols, and other advanced object-oriented features allow a smooth introduction to object-oriented programming to people with a background in Prolog programming. The distribution of Logtalk source code using an open-source license provide a framework for people to learn and then modify to try out new ideas on object-oriented programming research.

Structured knowledge representations and Knowledge-based systems: Logtalk objects, coupled with event-driven programming features, enable easy implementation of frame-like systems and similar structured knowledge representations.

Blackboard systems, Agent-based systems and systems with complex object relationships: Logtalk support for event-driven programming can provide a basis for the dynamic and reactive nature of these types of applications.

Highly portable applications: Logtalk is compatible with almost any modern Prolog compiler. Used as a way to provide Prolog with namespaces, it avoids the porting problems of most Prolog module systems. Platform, operating system, or compiler specific code can be isolated from the rest of the code by encapsulating it in objects with well-defined interfaces.

Alternative to a Prolog module system: Logtalk can be used as an alternative to a Prolog compiler module system. Any Prolog application that use modules can be converted to a Logtalk application, improving portability across Prolog compilers and taking advantage of the stronger reuse framework provided by Logtalk object-oriented features.

Integration with other programming languages: Logtalk support for most key object-oriented features helps users integrating Prolog with object-oriented languages like C++, Java, or Smalltalk by providing an high-level mapping between the two languages.

Writing programs

For a successful programming in Logtalk, you need a good working knowledge of Prolog and an understanding of the principles of object-oriented programming. All guidelines for writing good Prolog code apply as well to Logtalk programming. To those guidelines, you should add the basics of good object-oriented design.

One of the advantages of a system like Logtalk is that it enables the use of the currently available object-oriented methodologies, tools, and metrics [Champaux 92] in Prolog programming. That said, writing programs in Logtalk is similar to writing programs in Prolog: we define new predicates describing what is true about our domain objects, about our problem solution. We encapsulate our predicate directives and definitions inside new objects, categories and protocols that we create by hand with a text editor or by using the Logtalk built-in predicates. Some of the information collected during the analysis and design phases can

be integrated in the objects, categories and protocols that we define by using the available entity and predicate documenting directives.

Source files

A Logtalk source file must contain only one entity, either an object, a category, or a protocol. It is recommended that each source file be named after the entity identifier. For parametric objects, the identifier arity can be appended to the identifier functor. By default, all Logtalk source files use the extension `.lgt` but this is optional and can be set in the configuration files. Compiled source files (by the Logtalk pre-processor) have, by default, a `.pl` extension. Again, this can be set to match the needs of a particular Prolog compiler in the corresponding configuration file. For example, we may define an object named `vehicle` and save it in a `vehicle.lgt` source file that will be compiled to a `vehicle.pl` Prolog file. If we have a `sort(_)` parametric object we can save it on a `sort1.lgt` source file that will be compiled to a `sort1.pl` Prolog file. This name scheme helps avoid file name conflicts (remember that all Logtalk entities share the same name space).

Any Logtalk source file can contain arbitrary directives and clauses before the opening entity directive. These directives and clauses will not be compiled by the Logtalk pre-processor and will be copied unchanged to the beginning of the corresponding Prolog output file. This feature is included to help the integration of Logtalk with other Prolog extension like, for example, constraint programming extensions.

Avoiding common errors

Try to write objects and protocol documentation **before** writing any other code; if you are having trouble documenting a predicate, perhaps you need to go back to the design stage.

Try to avoid lengthy hierarchies. Besides performance penalties, composition is often a better choice over inheritance for defining new objects. In addition, prototype-based hierarchies are conceptually simpler and more efficient than class-based hierarchies.

Dynamic predicates or dynamic entities are sometimes needed, but we should always try to minimize the use of non-logical features such as destructive assignment (asserts and retracts).

Since each Logtalk entity is independently compiled, if an object inherits a dynamic or a metapredicate predicate, then we must repeat the respective directives in order to ensure a correct compilation.

In general, Logtalk does not verify if the predicate call/return arguments comply with the declared modes. On the other hand, Logtalk built-in predicates, built-in methods and message sending control structures are carefully checked for calling mode errors.

Logtalk error handling strongly depends on the ISO compliance of the chosen Prolog compiler. For instance, the error terms that are generated by some Logtalk built-in predicates assume that the Prolog built-ins behave as defined in the ISO standard regarding error conditions. In particular, if your Prolog compiler does not support a `read_term/3` built-in predicate compliant with the ISO Prolog Standard definition, then the current version of the Logtalk pre-processor will not be able to detect misspell variables in your source code.

Reference Manual

Grammar

The Logtalk grammar is here described using Backus-Naur Form syntax. Non-terminal symbols in *italics* have the definition found in the ISO Prolog Standard. Terminal symbols are represented in a `fixed width font` and between "".

Compilation units

```
entity ::=
      object |
      category |
      protocol
```

Object definition

object ::=
      begin_object_directive [object_directives] [clauses] end_object_directive.

begin_object_directive ::=
      `":- object("` object_identifier [`","` object_relations] `")."`

end_object_directive ::=
      `":- end_object."`

object_relations ::=
      prototype_relations |
      non_prototype_relations

prototype_relations ::=
      prototype_relation `","` prototype_relations |
      prototype_relation

prototype_relation ::=
      implements_protocols |
      imports_categories |
      extends_objects

non_prototype_relations ::=
      non_prototype_relation `","` non_prototype_relations |
      non_prototype_relation

non_prototype_relation ::=
      implements_protocols |
      imports_categories |
      instantiates_classes |
      specializes_classes

Category definition

category ::=
    begin_category_directive [category_directives] [clauses] end_category_directive.

begin_category_directive ::=
    ":- category(" category_identifier ["," implements_protocols] ")."

end_category_directive ::=
    ":- end_category."

Protocol definition

protocol ::=
    begin_protocol_directive [protocol_directives] end_protocol_directive.

begin_protocol_directive ::=
    ":- protocol(" protocol_identifier ["," extends_protocols] ")."

end_protocol_directive ::=
    ":- end_protocol."

Entity relations

implements_protocols ::=
    "implements(" implemented_protocols ")"

extends_protocols ::=
    "extends(" extended_protocols ")"

imports_categories ::=
    "imports(" imported_categories ")"

extends_objects ::=
    "extends(" extended_objects ")"

instantiates_classes ::=
    "instantiates(" instantiated_objects ")"

specializes_classes ::=
    "specializes(" specialized_objects ")"

Implemented protocols

implemented_protocols ::=
    implemented_protocol |
    implemented_protocol_sequence |
    implemented_protocol_list

implemented_protocol ::=
    protocol_identifier |
    entity_scope "::" protocol_identifier

implemented_protocol_sequence ::=
    implemented_protocol "," implemented_protocol_sequence |
    implemented_protocol

implemented_protocol_list ::=
    "[" implemented_protocol_sequence "]"

Extended protocols

extended_protocols ::=
    extended_protocol |
    extended_protocol_sequence |
    extended_protocol_list

extended_protocol ::=
    protocol_identifier |
    entity_scope ":" protocol_identifier

extended_protocol_sequence ::=
    extended_protocol "," extended_protocol_sequence |
    extended_protocol

extended_protocol_list ::=
    "[" extended_protocol_sequence "]"

Imported categories

imported_categories ::=
    imported_category |
    imported_category_sequence |
    imported_category_list

imported_category ::=
    category_identifier |
    entity_scope ":" category_identifier

imported_category_sequence ::=
    imported_category "," imported_category_sequence |
    imported_category

imported_category_list ::=
    "[" imported_category_sequence "]"

Extended objects

extended_objects ::=
    extended_object |
    extended_object_sequence |
    extended_object_list

extended_object ::=
    object_identifier |
    entity_scope ":" object_identifier

extended_object_sequence ::=
    extended_object "," extended_object_sequence |
    extended_object

extended_object_list ::=
"[" extended_object_sequence "]"

Instantiated objects

instantiated_objects ::=
instantiated_object |
instantiated_object_sequence |
instantiated_object_list

instantiated_object ::=
object_identifier |
entity_scope "::" object_identifier

instantiated_object_sequence ::=
instantiated_object "," instantiated_object_sequence |
instantiated_object

instantiated_object_list ::=
"[" instantiated_object_sequence "]"

Specialized objects

specialized_objects ::=
specialized_object |
specialized_object_sequence |
specialized_object_list

specialized_object ::=
object_identifier |
entity_scope "::" object_identifier

specialized_object_sequence ::=
specialized_object "," specialized_object_sequence |
specialized_object

specialized_object_list ::=
"[" specialized_object_sequence "]"

Entity scope

entity_scope ::=
"public" |
"protected" |
"private"

Entity identifiers

entity_identifiers ::=
entity_identifier |
entity_identifier_sequence |
entity_identifier_list

entity_identifier ::=
    object_identifier |
    protocol_identifier |
    category_identifier

entity_identifier_sequence ::=
    entity_identifier "," entity_identifier_sequence |
    entity_identifier

entity_identifier_list ::=
    "[" entity_identifier_sequence "]"

Object identifiers

object_identifiers ::=
    object_identifier |
    object_identifier_sequence |
    object_identifier_list

object_identifier ::=
    atom |
    compound

object_identifier_sequence ::=
    object_identifier "," object_identifier_sequence |
    object_identifier

object_identifier_list ::=
    "[" object_identifier_sequence "]"

Category identifiers

category_identifiers ::=
    category_identifier |
    category_identifier_sequence |
    category_identifier_list

category_identifier ::=
    atom

category_identifier_sequence ::=
    category_identifier "," category_identifier_sequence |
    category_identifier

category_identifier_list ::=
    "[" category_identifier_sequence "]"

Protocol identifiers

protocol_identifiers ::=
    protocol_identifier |
    protocol_identifier_sequence |
    protocol_identifier_list

protocol_identifier ::=
    atom

protocol_identifier_sequence ::=
    protocol_identifier "," protocol_identifier_sequence |
    protocol_identifier

protocol_identifier_list ::=
    "[" protocol_identifier_sequence "]"

Directives

Object directives

object_directives ::=
    object_directive object_directives |
    object_directive

object_directive ::=
    ":- initialization(" *callable* ")." |
    ":- uses(" object_identifiers ")." |
    ":- calls(" protocol_identifiers ")." |
    ":- dynamic." |
    ":- info(" info_list ")." |
    predicate_directives

Category directives

category_directives ::=
    category_directive category_directives |
    category_directive

category_directive ::=
    ":- initialization(" *callable* ")." |
    ":- uses(" object_identifiers ")." |
    ":- calls(" protocol_identifiers ")." |
    ":- dynamic." |
    ":- info(" info_list ")." |
    predicate_directives

Protocol directives

protocol_directives ::=
    protocol_directive protocol_directives |
    protocol_directive

protocol_directive ::=
    ":- initialization(" *callable* ")." |
    ":- dynamic." |
    ":- info(" info_list ")." |
    predicate_directives

Predicate directives

predicate_directives ::=
    predicate_directive predicate_directives |
    predicate_directive

predicate_directive ::=
    scope_directive |
    mode_directive |
    metapredicate_directive |
    info_directive |
    operator_directive |
    dynamic_directive |
    discontiguous_directive

scope_directive ::=
    ":- public(" predicate_indicator_term ")." |
    ":- protected(" predicate_indicator_term ")." |
    ":- private(" predicate_indicator_term ")."

mode_directive ::=
    ":- mode(" predicate_mode_term "," number_of_solutions ")."

metapredicate_directive ::=
    ":- metapredicate(" metapredicate_mode_indicator ")."

info_directive ::=
    ":- info(" predicate_indicator "," info_list ")."

predicate_indicator_term ::=
    predicate_indicator |
    predicate_indicator_sequence |
    predicate_indicator_list |

predicate_indicator_sequence ::=
    predicate_indicator "," predicate_indicator_sequence |
    predicate_indicator

predicate_indicator_list ::=
    "[" predicate_indicator_sequence "]"

predicate_mode_term ::=
    atom "(" mode_terms ")"
    mode_terms ::=
    mode_term "," mode_terms |
    mode_term
    mode_term ::=
    "@" [type] | "+" [type] | "-" [type] | "?" [type]

metapredicate_mode_indicator ::=
    atom "(" metapredicate_terms ")"

metapredicate_terms ::=
    metapredicate_term "," metapredicate_terms |
    metapredicate_term

metapredicate_term ::=
    "::" | "*"

type ::=
    prolog_type | logtalk_type | user_defined_type

prolog_type ::=
    "term" | "nonvar" | "var" |
    "compound" | "ground" | "callable" | "list" |
    "atomic" | "atom" |
    "number" | "integer" | "float"

logtalk_type ::=
    "object" | "category" | "protocol" |
    "event"

user_defined_type ::=
    atom |
    compound

number_of_solutions ::=
    "zero" | "zero_or_one" | "zero_or_more" | "one" | "one_or_more" | "error"

info_list ::=
    "[]" |
    "[" info_item "is" *nonvar* "|" info_list "]"

info_item ::=
    "comment" | "authors" | "version" | "date" | "parnames" |
    "argnames" | "definition" | "redefinition" | "allocation" |
    atom

Clauses

clauses ::=
    clause clauses |
    clause

goal ::=
    callable |
    message_call |
    external_call |
    built_in_method_call

message_call ::=
    message_to_object |
    message_to_self |
    message_to_super

message_to_object ::=
    receivers "::" messages

message_to_self ::=
    "::" messages

message_to_super ::=
    "^^" message

messages ::=
"(" message "," messages ")" |
"(" message ";" messages ")" |
message

message ::=
callable |
variable

receivers ::=
"(" receiver "," receivers ")" |
"(" receiver ";" receivers ")" |
receiver

receiver ::=
object_identifier |
variable

external_call ::=
"{" *callable* "}"

built_in_method_call ::=
"`self(`" *variable* "`)`" |
"`this(`" *variable* "`)`" |
"`sender(`" *variable* "`)`" |
"`parameter(`" *integer* "`,`" *variable* "`)`"

Entity properties

category_property ::=
"`static`" |
"`dynamic`" |
"`built_in`"

object_property ::=
"`static`" |
"`dynamic`" |
"`built_in`"

protocol_property ::=
"`static`" |
"`dynamic`" |
"`built_in`"

Predicate properties

predicate_property ::=
    "static" |
    "dynamic" |
    "private" |
    "protected" |
    "public" |
    "built_in" |
    "declared_in(" entity_identifier ")" |
    "defined_in(" object_identifier | category_identifier ")" |
    "metapredicate(" metapredicate_mode_indicator ")"

Directives

Entity directives

calls/1

Description

```
calls(Protocol)
calls(Protocol1, Protocol2, ...)
calls([Protocol1, Protocol2, ...])
```

Declares the protocol(s) that are called by predicates defined in an object or category.

Template and modes

```
calls(+protocol_identifiers)
```

Examples

```
:- calls(comparingp).
```

category/1-2

Description

```
category(Category)

category(Category,
      implements(Protocols))
```

Starting category directive.

Template and modes

```
category(+category_identifier)

category(+category_identifier,
      implements(+implemented_protocols))
```

Examples

```
:- category(monitoring).

:- category(monitoring,
      implements(monitoringp)).
```

```
:- category(attributes,
      implements(protected::variables)).
```

dynamic/0

Description

```
dynamic
```

Declares an entity and all of its directives and clauses dynamic.

Template and modes

```
dynamic
```

Examples

```
:- dynamic.
```

end_category/0

Description

```
end_category
```

Ending category directive.

Template and modes

```
end_category
```

Examples

```
:- end_category.
```

end_object/0

Description

```
end_object
```

Ending object directive.

Template and modes

```
end_object
```

Examples

```
:- end_object.
```

end_protocol/0

Description

```
end_protocol
```

Ending protocol directive.

Template and modes

```
end_protocol
```

Examples

```
:- end_protocol.
```

info/1

Description

```
info(List)
```

Documentation directive for objects, protocols, and categories.

Template and modes

```
info(+info_list)
```

Examples

```
:- info([
      version is 1.0,
      authors is 'Paulo Moura',
      date is 2000/4/20,
      comment is 'List protocol.']).
```

initialization/1

Description

```
initialization(Goal)
```

Sets a goal to be called immediately after the container entity has been loaded to memory.

Template and modes

```
initialization(@goal)
```

Examples

```
:- initialization(init).
```

object/1-4

Description

Stand-alone objects

```
object(Object)

object(Object,
      implements(Protocols))

object(Object,
      imports(Categories))

object(Object,
      implements(Protocols),
      imports(Categories))
```

Prototypes

```
object(Object,
      extends(Objects))

object(Object,
      implements(Protocols),
      extends(Objects))

object(Object,
      imports(Categories),
      extends(Objects))

object(Object,
      implements(Protocols),
      imports(Categories),
      extends(Objects))
```

Instances

```
object(Object,
      instantiates(Classes))

object(Object,
      implements(Protocols),
      instantiates(Classes))

object(Object,
      imports(Categories),
      instantiates(Classes))

object(Object,
      implements(Protocols),
      imports(Categories),
      instantiates(Classes))
```

Classes

```
object(Object,
      specializes(Classes))

object(Object,
      implements(Protocols),
      specializes(Classes))
```

```
object(Object,
      imports(Categories),
      specializes(Classes))

object(Object,
      implements(Protocols),
      imports(Categories),
      specializes(Classes))
```

Metaclasses

```
object(Object,
      instantiates(Classes),
      specializes(Classes))

object(Object,
      implements(Protocols),
      instantiates(Classes),
      specializes(Classes))

object(Object,
      imports(Categories),
      instantiates(Classes),
      specializes(Classes))

object(Object,
      implements(Protocols),
      imports(Categories),
      instantiates(Classes),
      specializes(Classes))
```

Starting object directive.

Template and modes

Stand-alone objects

```
object(+object_identifier)

object(+object_identifier,
      implements(+implemented_protocols))

object(+object_identifier,
      imports(+imported_categories))

object(+object_identifier,
      implements(+implemented_protocols),
      imports(+imported_categories))
```

Prototypes

```
object(+object_identifier,
      extends(+extended_objects))

object(+object_identifier,
      implements(+implemented_protocols),
      extends(+extended_objects))

object(+object_identifier,
      imports(+imported_categories),
      extends(+extended_objects))

object(+object_identifier,
      implements(+implemented_protocols),
      imports(+imported_categories),
      extends(+extended_objects))
```

Instances

```
object(+object_identifier,
      instantiates(+instantiated_objects))

object(+object_identifier,
      implements(+implemented_protocols),
      instantiates(+instantiated_objects))

object(+object_identifier,
      imports(+imported_categories),
      instantiates(+instantiated_objects))

object(+object_identifier,
      implements(+implemented_protocols),
      imports(+imported_categories),
      instantiates(+instantiated_objects))
```

Classes

```
object(+object_identifier,
      specializes(+specialized_objects))

object(+object_identifier,
      implements(+implemented_protocols),
      specializes(+specialized_objects))

object(+object_identifier,
      imports(+imported_categories),
      specializes(+specialized_objects))

object(+object_identifier,
      implements(+implemented_protocols),
      imports(+imported_categories),
      specializes(+specialized_objects))
```

Metaclasses

```
object(+object_identifier,
      instantiates(+instantiated_objects),
      specializes(+specialized_objects))

object(+object_identifier,
      implements(+implemented_protocols),
      instantiates(+instantiated_objects),
      specializes(+specialized_objects))

object(+object_identifier,
      imports(+imported_categories),
      instantiates(+instantiated_objects),
      specializes(+specialized_objects))

object(+object_identifier,
      implements(+implemented_protocols),
      imports(+imported_categories),
      instantiates(+instantiated_objects),
      specializes(+specialized_objects))
```

Examples

```
:- object(list).

:- object(list,
      implements(listp)).

:- object(list,
      extends(compound)).
```

```
:- object(list,
      implements(listp),
      extends(compound)).

:- object(object,
      imports(initialization),
      instantiates(class)).

:- object(abstract_class,
      instantiates(class),
      specializes(object)).

:- object(agent,
      imports(private::attributes)).
```

protocol/1-2

Description

```
protocol(Protocol)

protocol(Protocol,
      extends(Protocols))
```

Starting protocol directive.

Template and modes

```
protocol(+protocol_identifier)

protocol(+protocol_identifier,
      extends(+extended_protocols))
```

Examples

```
:- protocol(listp).

:- protocol(listp,
      extends(compoundp)).

:- protocol(queuep,
      extends(protected::listp)).
```

uses/1

Description

```
uses(Object)
uses(Object1, Object2, ...)
uses([Object1, Object2, ...])
```

Declares the object(s) that are sent messages by predicates defined in the category or object containing the directive.

Template and modes

```
uses(+object_identifiers)
```

Examples

```
:- uses(list).
```

Predicate directives

discontiguous/1

Description

```
discontiguous(Predicate)
discontiguous(Predicate1, Predicate2, ...)
discontiguous([Predicate1, Predicate2, ...])
```

Declares discontiguous predicates.

Template and modes

```
discontiguous(+predicate_indicator_term)
```

Examples

```
:- discontiguous(counter/1).

:- discontiguous(lives/2, works/2).

:- discontiguous([db/4, key/2, file/3]).
```

dynamic/1

Description

```
dynamic(Predicate)
dynamic(Predicate1, Predicate2, ...)
dynamic([Predicate1, Predicate2, ...])
```

Declares dynamic predicates. Note that an object can be static and have both static and dynamic predicates.

Template and modes

```
dynamic(+predicate_indicator_term)
```

Examples

```
:- dynamic(counter/1).

:- dynamic(lives/2, works/2).

:- dynamic([db/4, key/2, file/3]).
```

info/2

Description

```
info(Functor/Arity, List)
```

Documentation directive for predicates.

Template and modes

```
info(+predicate_indicator, +info_list)
```

Examples

```
:- info(empty/1, [
      comment is 'True if the argument is an empty list.',
      argnames is ['List']]).
```

metapredicate/1

Description

```
metapredicate(Metapredicate)
```

Declares metapredicates, i.e., predicates that have arguments that will be called as goals.

Template and modes

```
metapredicate(+metapredicate_predicate_term)
```

Examples

```
:- metapredicate(findall(*, ::, *)).

:- metapredicate(forall(::, ::)).
```

mode/2

Description

```
mode(Mode, Number_of_solutions)
```

Most predicates can be used with several instantiations modes. This directive enables the specification of each instantiation mode and the corresponding number of solutions/proofs.

Template and modes

```
mode(+predicate_mode_term, +number_of_solutions)
```

Examples

```
:- mode(append(-, -, +), zero_or_more).

:- mode(append(+list, +list, -list), zero_or_one).
```

```
:- mode(var(@term), zero_or_one).

:- mode(arg(-, -, +), error).
```

op/3

Description

```
op(Precedence, Associativity, Operator)
```

Declares operators.

Template and modes

```
op(+integer, +associativity, +atom)
```

Examples

```
:- op(950, fx, +).
:- op(950, fx, ?).
:- op(950, fx, @).
:- op(950, fx, -).
```

private/1

Description

```
private(Predicate)
private(Predicate1, Predicate2, ...)
private([Predicate1, Predicate2, ...])
```

Declares private predicates. A private predicate can only be called from the object containing the private directive.

Template and modes

```
private(+predicate_indicator_term)
```

Examples

```
:- private(counter/1).

:- private(init/1, free/1).

:- private([data/3, key/1, keys/1]).
```

protected/1

Description

```
protected(Predicate)
protected(Predicate1, Predicate2, ...)
protected([Predicate1, Predicate2, ...])
```

Declares protected predicates. A protected predicate can only be called from the object containing the declaration or from an object that inherits the declaration.

Template and modes

```
protected(+predicate_indicator_term)
```

Examples

```
:- protected(init/1).

:- protected(print/2, convert/4).

:- protected([load/1, save/3]).
```

public/1

Description

```
public(Predicate)
public(Predicate1, Predicate2, ...)
public([Predicate1, Predicate2, ...])
```

Declares public predicates. A public predicate can be called from any object.

Template and modes

```
public(+predicate_indicator_term)
```

Examples

```
:- public(ancestor/1).

:- public(instance/1, instances/1).

:- public([leaf/1, leaves/1]).
```

Built-in predicates

Enumerating objects, categories and protocols

current_category/1

Description

```
current_category(Category)
```

Enumerates, by backtracking, all currently defined categories. All categories are found, either static, dynamic, or built-in.

Template and modes

```
current_category(?category_identifier)
```

Errors

Category is neither a variable nor a valid category identifier:
`type_error(category_identifier, Category)`

Examples

```
| ?- current_category(monitoring).
```

current_object/1

Description

```
current_object(Object)
```

Enumerates, by backtracking, all currently defined objects. All objects are found, either static, dynamic or built-in.

Template and modes

```
current_object(?object_identifier)
```

Errors

Object is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`

Examples

```
| ?- current_object(list).
```

current_protocol/1

Description

```
current_protocol(Protocol)
```

Enumerates, by backtracking, all currently defined protocols. All protocols are found, either static, dynamic, or built-in.

Template and modes

```
current_protocol(?protocol_identifier)
```

Errors

Protocol is neither a variable nor a valid protocol identifier:
`type_error(protocol_identifier, Protocol)`

Examples

```
| ?- current_protocol(listp).
```

Enumerating objects, categories and protocols properties

category_property/2

Description

```
category_property(Category, Property)
```

Enumerates, by backtracking, the properties associated with the defined categories.

Template and modes

```
category_property(?category_identifier, ?category_property)
```

Errors

Category is neither a variable nor a valid category identifier:
`type_error(category_identifier, Category)`
Property is neither a variable nor a valid category property:
`domain_error(category_property, Property)`

Examples

```
| ?- category_property(Category, dynamic).
```

object_property/2

Description

```
object_property(Object, Property)
```

Enumerates, by backtracking, the properties associated with the defined objects.

Template and modes

```
object_property(?object_identifier, ?object_property)
```

Errors

Object is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`
Property is neither a variable nor a valid object property:
`domain_error(object_property, Property)`

Examples

```
| ?- object_property(list, Property).
```

protocol_property/2

Description

```
protocol_property(Protocol, Property)
```

Enumerates, by backtracking, the properties associated with the currently defined protocols.

Template and modes

```
protocol_property(?protocol_identifier, ?protocol_property)
```

Errors

Protocol is neither a variable nor a valid protocol identifier:
`type_error(protocol_identifier, Protocol)`
Property is neither a variable nor a valid protocol property:
`domain_error(protocol_property, Property)`

Examples

```
| ?- protocol_property(listp, Property).
```

Creating new objects, categories and protocols

create_category/4

Description

```
create_category(Identifier, Relations, Directives, Clauses)
```

Creates a new, dynamic, category.

Template and modes

```
create_category(+category_identifier, +list, +list, +list)
```

Errors

Identifier is a variable:
```
instantiation_error
```
Identifier is not a valid category identifier:
```
type_error(category_identifier, Identifier)
```
Identifier is already in use:
```
permission_error(replace, category, Identifier)
permission_error(replace, object, Identifier)
permission_error(replace, protocol, Identifier)
```
Relations is not a list:
```
type_error(list, Relations)
```
Directives is not a list:
```
type_error(list, Directives)
```
Clauses is not a list:
```
type_error(list, Clauses)
```

Examples

```
| ?- create_category(foo, [implements(bar)], [], [bar(1), bar(2)]).
```

create_object/4

Description

```
create_object(Identifier, Relations, Directives, Clauses)
```

Creates a new, dynamic, object.

Template and modes

```
create_object(+object_identifier, +list, +list, +list)
```

Errors

Identifier is a variable:
```
instantiation_error
```
Identifier is not a valid object identifier:
```
type_error(object_identifier, Identifier)
```

Identifier is already in use:
```
permission_error(replace, category, Identifier)
permission_error(replace, object, Identifier)
permission_error(replace, protocol, Identifier)
```
Relations is not a list:
```
type_error(list, Relations)
```
Directives is not a list:
```
type_error(list, Directives)
```
Clauses is not a list:
```
type_error(list, Clauses)
```

Examples

```
| ?- create_object(foo, [extends(bar)], [public(foo/1)], [foo(1), foo(2)]).
```

create_protocol/3

Description

```
create_protocol(Identifier, Relations, Directives)
```

Creates a new, dynamic, protocol.

Template and modes

```
create_protocol(+protocol_identifier, +list, +list)
```

Errors

Identifier is a variable:
```
instantiation_error
```
Identifier is not a valid protocol identifier:
```
type_error(protocol_identifier, Identifier)
```
Identifier is already in use:
```
permission_error(replace, category, Identifier)
permission_error(replace, object, Identifier)
permission_error(replace, protocol, Identifier)
```
Relations is not a list:
```
type_error(list, Relations)
```
Directives is not a list:
```
type_error(list, Directives)
```

Examples

```
| ?- create_protocol(foo, [extends(bar)], [public(foo/1)]).
```

Abolishing objects, categories and protocols

abolish_category/1

Description

```
abolish_category(Category)
```

Abolishes from the database a dynamic category.

Template and modes

```
abolish_category(@category_identifier)
```

Errors

Category is a variable:
```
instantiation_error
```
Category is not a valid category identifier:
```
type_error(category_identifier, Category)
```
Category is an identifier of a static category:
```
permission_error(modify, static_category, Category)
```
Category does not exist:
```
existence_error(category, Category)
```

Examples

```
| ?- abolish_category(monitoring).
```

abolish_object/1

Description

```
abolish_object(Object)
```

Abolishes from the database a dynamic object.

Template and modes

```
abolish_object(@object_identifier)
```

Errors

Object is a variable:
```
instantiation_error
```
Object is not a valid object identifier:
```
type_error(object_identifier, Object)
```
Object is an identifier of a static object:
```
permission_error(modify, static_object, Object)
```
Object does not exist:
```
existence_error(object, Object)
```

Examples

```
| ?- abolish_object(list).
```

abolish_protocol/1

Description

```
abolish_protocol(Protocol)
```

Abolishes from the database a dynamic protocol.

Template and modes

```
abolish_protocol(@protocol_identifier)
```

Errors

Protocol is a variable:
`instantiation_error`
Protocol is not a valid protocol identifier:
`type_error(protocol_identifier, Protocol)`
Protocol is an identifier of a static protocol:
`permission_error(modify, static_protocol, Protocol)`
Protocol does not exist:
`existence_error(protocol, Protocol)`

Examples

```
| ?- abolish_protocol(listp).
```

Object, category, and protocol relations

extends_object/2-3

Description

```
extends_object(Prototype, Parent)

extends_object(Prototype, Parent, Scope)
```

Enumerates, by backtracking, all pairs of objects such that the first one extends the second.

Template and modes

```
extends_object(?object_identifier, ?object_identifier)

extends_object(?object_identifier, ?object_identifier, ?entity_scope)
```

Errors

Prototype is neither a variable nor a valid object identifier:
`type_error(object_identifier, Prototype)`

Parent is neither a variable nor a valid object identifier:
```
type_error(object_identifier, Parent)
```
Scope is neither a variable nor a valid entity scope:
```
type_error(entity_scope, Scope)
```

Examples

```
| ?- extends_object(Object, state_space).

| ?- extends_object(Object, list, public).
```

extends_protocol/2-3

Description

```
extends_protocol(Protocol1, Protocol2)

extends_protocol(Protocol1, Protocol2, Scope)
```

Enumerates, by backtracking, all pairs of protocols such that the first one extends the second.

Template and modes

```
extends_protocol(?protocol_identifier, ?protocol_identifier)

extends_protocol(?protocol_identifier, ?protocol_identifier, ?entity_scope)
```

Errors

Protocol1 is neither a variable nor a valid protocol identifier:
```
type_error(protocol_identifier, Protocol1)
```
Protocol2 is neither a variable nor a valid protocol identifier:
```
type_error(protocol_identifier, Protocol2)
```
Scope is neither a variable nor a valid entity scope:
```
type_error(entity_scope, Scope)
```

Examples

```
| ?- extends_protocol(listp, Protocol).

| ?- extends_protocol(Protocol, termp, private).
```

implements_protocol/2-3

Description

```
implements_protocol(Object, Protocol)
implements_protocol(Category, Protocol)

implements_protocol(Object, Protocol, Scope)
implements_protocol(Category, Protocol, Scope)
```

Enumerates, by backtracking, all pairs of entities such that an object or a category implements a protocol.

Template and modes

```
implements_protocol(?object_identifier, ?protocol_identifier)
implements_protocol(?category_identifier, ?protocol_identifier)
```

```
implements_protocol(?object_identifier, ?protocol_identifier, ?entity_scope)
implements_protocol(?category_identifier, ?protocol_identifier, ?entity_scope)
```

Errors

Object is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`
Category is neither a variable nor a valid category identifier:
`type_error(category_identifier, Category)`
Protocol is neither a variable nor a valid protocol identifier:
`type_error(protocol_identifier, Protocol)`
Scope is neither a variable nor a valid entity scope:
`type_error(entity_scope, Scope)`

Examples

```
| ?- implements_protocol(List, listp).

| ?- implements_protocol(List, listp, public).
```

imports_category/2-3

Description

```
imports_category(Object, Category)

imports_category(Object, Category, Scope)
```

Enumerates, by backtracking, all pairs of objects and categories such that the first one imports the other.

Template and modes

```
imports_category(?object_identifier, ?category_identifier)

imports_category(?object_identifier, ?category_identifier, ?entity_scope)
```

Errors

Object is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`
Category is neither a variable nor a valid category identifier:
`type_error(category_identifier, Category)`
Scope is neither a variable nor a valid entity scope:
`type_error(entity_scope, Scope)`

Examples

```
| ?- imports_category(debugger, monitoring).

| ?- imports_category(Object, monitoring, protected).
```

instantiates_class/2-3

Description

```
instantiates_class(Instance, Class)

instantiates_class(Instance, Class, Scope)
```

Enumerates, by backtracking, all pairs of objects such that the first one instantiates the second.

Template and modes

```
instantiates_class(?object_identifier, ?object_identifier)

instantiates_class(?object_identifier, ?object_identifier, ?entity_scope)
```

Errors

Instance is neither a variable nor a valid object identifier:
`type_error(object_identifier, Instance)`
Class is neither a variable nor a valid object identifier:
`type_error(object_identifier, Class)`
Scope is neither a variable nor a valid entity scope:
`type_error(entity_scope, Scope)`

Examples

```
| ?- instantiates_class(water_jug, state_space).

| ?- instantiates_class(Space, state_space, public).
```

specializes_class/2-3

Description

```
specializes_class(Class, Superclass)

specializes_class(Class, Superclass, Scope)
```

Enumerates, by backtracking, all pairs of objects such that the first one specializes the second.

Template and modes

```
specializes_class(?object_identifier, ?object_identifier)

specializes_class(?object_identifier, ?object_identifier, ?entity_scope)
```

Errors

Class is neither a variable nor a valid object identifier:
`type_error(object_identifier, Class)`
Superclass is neither a variable nor a valid object identifier:
`type_error(object_identifier, Superclass)`
Scope is neither a variable nor a valid entity scope:
`type_error(entity_scope, Scope)`

Examples

```
| ?- specializes_class(Subclass, state_space).

| ?- specializes_class(Subclass, state_space, public).
```

Event handling

abolish_events/5

Description

```
abolish_events(Event, Object, Message, Sender, Monitor)
```

Abolishes all matching events.

Template and modes

```
abolish_events(@event, @object_identifier, @callable, @object_identifier, @object_identifier)
```

Errors

Event is neither a variable nor a valid event identifier:
`type_error(event, Event)`
Object is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`
Message is neither a variable nor a callable term:
`type_error(callable, Message)`
Sender is neither a variable nor a valid object identifier:
`type_error(object_identifier, Sender)`
Monitor is neither a variable nor a valid object identifier:
`type_error(object_identifier, Monitor)`

Examples

```
| ?- abolish_events(_, list, _, _, debugger).
```

current_event/5

Description

```
current_event(Event, Object, Message, Sender, Monitor)
```

Enumerates, by backtracking, all defined events.

Template and modes

```
current_event(?event, ?object_identifier, ?callable, ?object_identifier, ?object_identifier)
```

Errors

Event is neither a variable nor a valid event identifier:
`type_error(event, Event)`

Object is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`
Message is neither a variable nor a callable term:
`type_error(callable, Message)`
Sender is neither a variable nor a valid object identifier:
`type_error(object_identifier, Sender)`
Monitor is neither a variable nor a valid object identifier:
`type_error(object_identifier, Monitor)`

Examples

```
| ?- current_event(Event, Object, Message, Sender, debugger).
```

define_events/5

Description

```
define_events(Event, Object, Message, Sender, Monitor)
```

Defines a new set of events.

Template and modes

```
define_events(@event, @object_identifier, @callable, @object_identifier, +object_identifier)
```

Errors

Event is neither a variable nor a valid event identifier:
`type_error(event, Event)`
Object is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`
Message is neither a variable nor a callable term:
`type_error(callable, Message)`
Sender is neither a variable nor a valid object identifier:
`type_error(object_identifier, Sender)`
Monitor is a variable:
`instantiation_error`
Monitor is neither a variable nor a valid object identifier:
`type_error(object_identifier, Monitor)`

Examples

```
| ?- define_events(_, list, member(_, _), _ , debugger).
```

Compiling and loading objects, categories, and protocols

logtalk_compile/1

Description

```
logtalk_compile(Entity)
```

Compiles to disk a Logtalk entity or a list of entities (objects, protocols, or categories). The Logtalk file name extension (by default, `.lgt`) should be omitted.

Template and modes

```
logtalk_compile(+atom)
logtalk_compile(+atom_list)
```

Errors

Entity is a variable:
`instantiation_error`
Entity is neither a variable nor an atom:
`type_error(atom, Entity)`
Entity does not exist:
`existence_error(entity, Entity)`

Examples

```
| ?- logtalk_compile(list).

| ?- logtalk_compile([listp, list]).
```

logtalk_load/1

Description

```
logtalk_load(Entity)
```

Compiles to disk and then loads to memory a Logtalk entity or a list of entities (objects, protocols or categories). The Logtalk file name extension (by default, `.lgt`) should be omitted.

Template and modes

```
logtalk_load(+atom)
logtalk_load(+atom_list)
```

Errors

Entity is a variable:
`instantiation_error`
Entity is neither a variable nor an atom:
`type_error(atom, Entity)`
Entity does not exist:
`existence_error(entity, Entity)`

Examples

```
| ?- logtalk_load(list).

| ?- logtalk_load([listp, list]).
```

Others

forall/2

Description

```
forall(Generator, Test)
```

This predicate is true if, for all solutions of Generator, Test is true (some Prolog compilers already define this or a similar predicate).

Template and modes

```
forall(+callable, +callable)
```

Errors

Generator is not a callable term:
```
type_error(callable, Generator)
```
Test is not a callable term:
```
type_error(callable, Test)
```

Examples

```
| ?- forall(member(X, [1, 2, 3]), write(X)).
```

logtalk_version/3

Description

```
logtalk_version(Major, Minor, Patch)
```

Returns the Logtalk pre-processor and runtime version.

Template and modes

```
logtalk_version(?integer, ?integer, ?integer)
```

Errors

Major is neither a variable nor an integer:
```
type_error(integer, Major)
```
Minor is neither a variable nor an integer:
```
type_error(integer, Minor)
```
Patch is neither a variable nor an integer:
```
type_error(integer, Patch)
```

Examples

```
| ?- logtalk_version(Major, Minor, Patch).
```

retractall/1

Description

```
retractall(Head)
```

Logtalk adds this built-in predicate, with the usual definition, to a Prolog compiler if it is not already defined.

Template and modes

```
retractall(+callable)
```

Errors

Head is not a callable term:
```
type_error(callable, Head)
```

Examples

```
| ?- retractall(foo(_)).
```

Built-in methods

Local methods

parameter/2

Description

```
parameter(Number, Term)
```

Used only in *parametric objects*, this method returns parameter values by using the parameter position in the entity identifier. See also `this/1`.

Template and modes

```
parameter(+integer, ?term)
```

Errors

Number is a variable:
`instantiation_error`
Number is neither a variable nor an integer value:
`type_error(integer, Number)`
Object identifier is not a compound term:
`type_error(compound, Object)`
Number is a negative integer value:
`domain_error(not_less_than_zero, Number)`

Examples

```
:- object(box(_Colour)).

      ...

      colour(Colour) :-
            parameter(1, Colour).

      ...
```

self/1

Description

```
self(Self)
```

Returns the object that has received the message under processing.

Template and modes

```
self(-object)
```

Errors

(none)

Examples

```
test :-
      self(Self),
      write('executing a message in behalf of '),
      writeq(Self), nl.
```

sender/1

Description

```
sender(Sender)
```

Returns the object that has sent the message under processing.

Template and modes

```
sender(-object)
```

Errors

(none)

Examples

```
test :-
      sender(Sender),
      write('executing a message sent by '),
      writeq(Sender), nl.
```

this/1

Description

```
this(This)
```

Returns the object that contains the predicate definition that is being executed. This method is useful in avoiding problems when an object is renamed or when using parametric objects. Can also be used to retrieve runtime parametric object parameters though unification (see also `parameter/2`).

Template and modes

```
this(-object_identifier)
```

Errors

(none)

Examples

```
test :-
      this(This),
      write('executing a definition contained in '),
      writeq(This), nl.
```

Reflection methods

current_predicate/1

Description

```
current_predicate(Predicate)
```

Enumerates, by backtracking, the visible user predicates for an object.

Template and modes

```
current_predicate(?predicate_indicator)
```

Errors

Predicate is neither a variable nor a valid predicate indicator:
`type_error(predicate_indicator, Predicate)`

Examples

To enumerate, by backtracking, the user predicates visible in *this*:
`current_predicate(Predicate)`
To enumerate, by backtracking, the public and protected user predicates visible in *self*:
`::current_predicate(Predicate)`
To enumerate, by backtracking, the public user predicates visible for an object:
`Object::current_predicate(Predicate)`

predicate_property/2

Description

```
predicate_property(Predicate, Property)
```

Enumerates, by backtracking, the properties of a visible predicate.

Template and modes

```
predicate_property(+callable, ?predicate_property)
```

Errors

Predicate is a variable:
`instantiation_error`
Predicate is neither a variable nor a callable term:
`type_error(callable, Predicate)`

Property is neither a variable nor a valid predicate property:
```
domain_error(predicate_property, Property)
```

Examples

To enumerate, by backtracking, the properties of a predicate visible in *this*:
```
predicate_property(foo(_), Property)
```
To enumerate, by backtracking, the properties of a public or protected predicate visible in *self*:
```
::predicate_property(foo(_), Property)
```
To enumerate, by backtracking, the properties of a public predicate visible in an object:
```
Object::predicate_property(foo(_), Property)
```

Database methods

abolish/1

Description

```
abolish(Predicate)
abolish(Functor/Arity)
```

Removes a dynamic predicate from an object database. Note however that if the dynamic predicate is declared in a category the predicate will fail.

Template and modes

```
abolish(+predicate_indicator)
```

Errors

Predicate is a variable:
```
instantiation_error
```
Predicate is neither a variable nor a valid predicate indicator:
```
type_error(predicate_indicator, Predicate)
```
Functor is neither a variable nor an atom:
```
type_error(atom, Functor)
```
Arity is neither a variable nor an integer:
```
type_error(integer, Arity)
```
Predicate is statically declared:
```
permission_error(modify, predicate_declaration, Functor/Arity)
```
Predicate is a private predicate:
```
permission_error(modify, private_predicate, Functor/Arity)
```
Predicate is a protected predicate:
```
permission_error(modify, protected_predicate, Functor/Arity)
```
Predicate is a static predicate:
```
permission_error(modify, static_predicate, Functor/Arity)
```
Predicate is not declared:
```
existence_error(predicate_declaration, Functor/Arity)
```

Examples

To abolish any dynamic predicate in *this*:
```
abolish(Predicate)
```
To abolish a public or protected dynamic predicate in *self*:
```
::abolish(Predicate)
```

To abolish a public dynamic predicate in an object:
```
Object::abolish(Predicate)
```

asserta/1

Description

```
asserta(Clause)
asserta((Head:-Body))
```

Asserts a clause as the first one for an object's dynamic predicate. If the predicate is not already declared, then a dynamic predicate declaration is added to the object.

Template and modes

```
asserta(+clause)
```

Errors

Clause is a variable:
```
instantiation_error
```
Head is a variable:
```
instantiation_error
```
Head is a neither a variable nor a callable term:
```
type_error(callable, Head)
```
Body cannot be converted to a goal:
```
type_error(callable, Body)
```
The predicate indicator of Head is that of a private predicate:
```
permission_error(modify, private_predicate, Head)
```
The predicate indicator of Head is that of a protected predicate:
```
permission_error(modify, protected_predicate, Head)
```
The predicate indicator of Head is that of a static predicate:
```
permission_error(modify, static_predicate, Head)
```

Examples

To assert a clause as the first one for any dynamic predicate in *this*:
```
asserta(Clause)
```
To assert a clause as the first one for any public or protected dynamic predicate in *self*:
```
::asserta(Clause)
```
To assert a clause as the first one for any public dynamic predicate in an object:
```
Object::asserta(Clause)
```

assertz/1

Description

```
assertz(Clause)
assertz((Head:-Body))
```

Asserts a clause as the last one for an object's dynamic predicate. If the predicate is not already declared, then a dynamic predicate declaration is added to the object.

Template and modes

```
assertz(+clause)
```

Errors

Clause is a variable:
```
instantiation_error
```
Head is a variable:
```
instantiation_error
```
Head is a neither a variable nor a callable term:
```
type_error(callable, Head)
```
Body cannot be converted to a goal:
```
type_error(callable, Body)
```
The predicate indicator of Head is that of a private predicate:
```
permission_error(modify, private_predicate, Head)
```
The predicate indicator of Head is that of a protected predicate:
```
permission_error(modify, protected_predicate, Head)
```
The predicate indicator of Head is that of a static predicate:
```
permission_error(modify, static_predicate, Head)
```

Examples

To assert a clause as the last one for any dynamic predicate in *this*:
```
assertz(Clause)
```
To assert a clause as the last one for any public or protected dynamic predicate in *self*:
```
::assertz(Clause)
```
To assert a clause as the last one for any public dynamic predicate in an object:
```
Object::assertz(Clause)
```

clause/2

Description

```
clause(Head, Body)
```

Enumerates, by backtracking, the clauses of an object's dynamic predicates. Note however that if the clauses for the dynamic predicate are contained in a category the predicate will fail.

Template and modes

```
clause(?callable, ?body)
```

Errors

Head is a variable:
```
instantiation_error
```
Head is a neither a variable nor a callable term:
```
type_error(callable, Head)
```
Body is a neither a variable nor a callable term:
```
type_error(callable, Body)
```
The predicate indicator of Head is that of a private predicate:
```
permission_error(access, private_predicate, Head)
```
The predicate indicator of Head is that of a protected predicate:
```
permission_error(access, protected_predicate, Head)
```
The predicate indicator of Head is that of a static predicate:
```
permission_error(access, static_predicate, Head)
```
Head is not a declared predicate:
```
existence_error(predicate_declaration, Head)
```

Examples

To retrieve a matching clause of any dynamic predicate in *this*:
```
clause(Head, Body)
```
To retrieve a matching clause of a public or protected dynamic predicate in *self*:
```
::clause(Head, Body)
```
To retrieve a matching clause of a public dynamic predicate in an object:
```
Object::clause(Head, Body)
```

retract/1

Description

```
retract(Clause)
retract((Head:-Body))
```

Retracts a dynamic clause from an object. Note however that if the clauses for the dynamic predicate are contained in a category the predicate will fail.

Template and modes

```
retract(+clause)
```

Errors

Head is a variable:
```
instantiation_error
```
Head is neither a variable nor a callable term:
```
type_error(callable, Head)
```
The predicate indicator of Head is that of a private predicate:
```
permission_error(modify, private_predicate, Head)
```
The predicate indicator of Head is that of a protected predicate:
```
permission_error(modify, protected_predicate, Head)
```
The predicate indicator of Head is that of a static predicate:
```
permission_error(modify, static_predicate, Head)
```
The predicate indicator of Head is not declared:
```
existence_error(predicate_declaration, Head)
```

Examples

To retract a matching clause of any dynamic predicate in *this*:
```
retract(Clause)
```
To retract a matching clause of a public or protected dynamic predicate in *self*:
```
::retract(Clause)
```
To retract a matching clause of a public dynamic predicate in an object:
```
Object::retract(Clause)
```

retractall/1

Description

```
retractall(Head)
```

Retracts all matching predicates from an object. Note however that if the clauses for the dynamic predicate are contained in a category the predicate will fail.

Template and modes

```
retractall(+callable)
```

Errors

Head is a variable:
`instantiation_error`
Head is neither a variable nor a callable term:
`type_error(callable, Head)`
The predicate indicator of Head is that of a private predicate:
`permission_error(modify, private_predicate, Head)`
The predicate indicator of Head is that of a protected predicate:
`permission_error(modify, protected_predicate, Head)`
The predicate indicator of Head is that of a static predicate:
`permission_error(modify, static_predicate, Head)`
The predicate indicator of Head is not declared:
`existence_error(predicate_declaration, Head)`

Examples

To retract all matching predicate definitions in *this*:
`retractall(Head)`
To retract all matching public or protected predicate definitions in *self*:
`::retractall(Head)`
To retract all matching public predicate definitions in an object:
`Object::retractall(Head)`

All solutions methods

bagof/3

Description

```
bagof(Term, Goal, List)
```

(See the Prolog ISO standard definition)

Template and modes

```
bagof(@term, +callable, -list)
```

Errors

(See the Prolog ISO standard)

Examples

To find all solutions in *this*:
`bagof(Term, Goal, List)`
To find all solutions in *self*:
`bagof(Term, ::Goal, List)`
To find all solutions in an object:
`bagof(Term, Object::Goal, List)`

findall/3

Description

```
findall(Term, Goal, List)
```

(See the Prolog ISO standard definition)

Template and modes

```
findall(@term, +callable, -list)
```

Errors

(See the Prolog ISO standard)

Examples

To find all solutions in *this*:
```
findall(Term, Goal, List)
```
To find all solutions in *self*:
```
findall(Term, ::Goal, List)
```
To find all solutions in an object:
```
findall(Term, Object::Goal, List)
```

forall/2

Description

```
forall(Generator, Test)
```

For all solutions of `Generator`, `Test` is true.

Template and modes

```
forall(+callable, +callable)
```

Errors

Either Generator or Test is a variable:
```
instantiation_error
```
Generator is neither a variable nor a callable term:
```
type_error(callable, Generator)
```
Test is neither a variable nor a callable term:
```
type_error(callable, Test)
```

Examples

To call both goals in *this*:
```
forall(Generator, Test)
```
To call both goals in *self*:
```
forall(::Generator, ::Test)
```
To call both goals in an object:
```
forall(Object::Generator, Object::Test)
```

setof/3

Description

```
setof(Term, Goal, List)
```

(See the Prolog ISO standard definition)

Template and modes

```
setof(@term, +callable, -list)
```

Errors

(See the Prolog ISO standard)

Examples

To find all solutions in *this*:
```
setof(Term, Goal, List)
```
To find all solutions in *self*:
```
setof(Term, ::Goal, List)
```
To find all solutions in an object:
```
setof(Term, Object::Goal, List)
```

Event handler methods

before/3

Description

```
before(Object, Message, Sender)
```

This is a pre-declared but user-defined public method for handling `before` events.

Template and modes

```
before(?object, ?term, ?object)
```

Errors

(none)

Examples

```
before(Object, Message, Sender) :-
      writeq(Object), write('::'), writeq(Message),
      write(' from '), writeq(Sender), nl.
```

after/3

Description

```
after(Object, Message, Sender)
```

This is a pre-declared but user-defined public method for handling `after` events.

Template and modes

```
after(?object, ?term, ?object)
```

Errors

(none)

Examples

```
after(Object, Message, Sender) :-
      writeq(Object), write('::'), writeq(Message),
      write(' from '), writeq(Sender), nl.
```

Control constructs

Message sending

::/2

Description

```
Object::Predicate

(Object1, Object2, ...)::Predicate

(Object1; Object2, ...)::Predicate

Object::(Predicate1, Predicate2, ...)

Object::(Predicate1; Predicate2; ...)
```

Sends a message to an object. The message argument must match a public predicate of the receiver object.

Template and modes

```
+receivers::+messages
```

Errors

Either Object or Predicate is a variable:
`instantiation_error`
Predicate is declared private:
`permission_error(access, private_predicate, Predicate)`
Predicate is declared protected:
`permission_error(access, protected_predicate, Predicate)`
Predicate is not declared:
`existence_error(predicate_declaration, Predicate)`
Object does not exist:
`existence_error(object, Object)`

Examples

```
| ?- list::member(X, [1, 2, 3]).
```

::/1

Description

```
::Predicate

::(Predicate1, Predicate2, ...)

::(Predicate1; Predicate2; ...)
```

Send a message to *self*. Only used in the body of a predicate definition. The argument should match a public or protected predicate of *self*. It may also match a private predicate if the predicate is imported from a category, if used from inside a category, or when using private inheritance.

Template and modes

```
::+messages
```

Errors

Predicate is a variable:
`instantiation_error`
Predicate is declared private:
`permission_error(access, private_predicate, Predicate)`
Predicate is not declared:
`existence_error(predicate_declaration, Predicate)`

Examples

```
area(Area) :-
      ::width(Width),
      ::height(Height),
      Area is Width*Height.
```

^^/1

Description

```
^^Predicate
```

Calls a redefined/inherited definition for a message. Normally only used in the body of a predicate definition for the message. Predicate should match a public or protected predicate of *self* or be within the scope of *this*.

Template and modes

```
^^+message
```

Errors

Predicate is a variable:
`instantiation_error`
Predicate is declared private:
`permission_error(access, private_predicate, Predicate)`
Predicate is not declared:
`existence_error(predicate_declaration, Predicate)`
Container of the inherited predicate definition is the same object that contains the `^^/1` call:
`endless_loop(Predicate)`

Examples

```
init :-
      assertz(counter(0)),
      ^^init.
```

Calling external code

{}/1

Description

```
{Goal}
```

Calls external Prolog code. Can be used to bypass the Logtalk pre-processor.

Template and modes

```
{+callable}
```

Errors

(none)

Examples

```
N1/D1 < N2/D2 :-
      {N1*D2 < N2*D1}.
```

Tutorial

List predicates

In this example, we will illustrate the use of:

- objects
- protocols

using common list utility predicates.

Defining a list object

We will start by defining an object, `list`, containing predicate definitions for some common list predicates like `append/3`, `length/2` and `member/2`:

```
:- object(list).

      :- public(append/3).
      :- public(length/2).
      :- public(member/2).

      append([], List, List).

      append([Head| Tail], List, [Head| Tail2]) :-
            append(Tail, List, Tail2).

      length(List, Length) :-
            length(List, 0, Length).

      length([], Length, Length).

      length([_| Tail], Acc, Length) :-
            Acc2 is Acc + 1,
            length(Tail, Acc2, Length).

      member(Element, [Element| _]).

      member(Element, [_| List]) :-
            member(Element, List).

:- end_object.
```

What is different here from a regular Prolog program? The definitions of the list predicates are the usual ones. We have two new directives, `object/1` and `end_object/0`, that encapsulate the object's code. In Logtalk, by default, all object predicates are private; therefore, we have to explicitly declare all predicates that we want to be public, that is, that we want to call from outside the object. This is done using the `public/1` scope directive.

After compiling and loading this new object, we can now try goals like:

```
| ?- list::member(X, [1, 2, 3]).

X = 1;
X = 2;
X = 3;
no
```

Or:

```
| ?- list::length([1, 2, 3], L).

L = 3
yes
```

The infix operator `::/2` is used in Logtalk to send a message to an object.

Defining a list protocol

As we saw in the above example, a Logtalk object may contain predicate directives and predicate definitions. The set of predicate directives defines what we call the object's *protocol* or interface. An interface may have several implementations. For instance, we may want to define a new object that implements the list predicates using difference lists. However, we do not want to repeat the predicate directives in the new object. Therefore, what we need is to split the object's protocol from the object's predicate definitions by defining a new Logtalk entity called a protocol. Logtalk protocols are compilations units, at the same level as objects and categories. That said, let us define a `listp` protocol:

```
:- protocol(listp).

      :- public(append/3).
      :- public(length/2).
      :- public(member/2).

:- end_protocol.
```

Similar to what we have done for objects, we use the directives `protocol/1` and `end_protocol/0` to encapsulate the predicate directives. We can improve this protocol by documenting the call/return modes and the number of solutions of each predicate using the `mode/2` directive:

```
:- protocol(listp).

      :- public(append/3).
      :- mode(append(?list, ?list, ?list), zero_or_more).

      :- public(length/2).
      :- mode(length(?list, ?integer), zero_or_more).

      :- public(member/2).
      :- mode(member(?term, ?list), zero_or_more).

:- end_protocol.
```

Now we need to change our definition of the `list` object. We remove the predicate directives and state that the object implements the `listp` protocol:

```
:- object(list,
      implements(listp)).

      append([], List, List).

      append([Head| Tail], List, [Head| Tail2]) :-
            append(Tail, List, Tail2).

      ...

:- end_object.
```

The protocol declared in `listp` may now be alternatively implemented using difference lists by defining a new object, `difflist`:

```
:- object(difflist,
      implements(listp).

      append(L1-X, X-L2, L1-L2).

      ...

:- end_object.
```

Summary

- It is easy to define a simple object: just put your Prolog code inside starting and ending object directives and add the necessary scope directives. The object will be self-defining and ready to use.

- Define a protocol when you may want to provide or enable several alternative definitions to a given set of predicates. This way we avoid needless repetition of predicate directives.

Dynamic object attributes

In this example, we will illustrate the use of:

- categories
- category predicates
- dynamic predicates

by defining a category that implements a set of predicates for handling dynamic object attributes.

Defining a category

We want to define a set of predicates to handle dynamic object attributes. We need public predicates to set, get, and delete attributes, and a private dynamic predicate to store the attributes values. Let us name these predicates `set_attribute/2` and `get_attribute/2`, for getting and setting an attribute value, `del_attribute/2` and `del_attributes/2`, for deleting attributes, and `attribute_/2`, for storing the attributes values.

We do not want to encapsulate these predicates in an object. Why? Because they are a set of useful, closely related predicates that may be used by several unrelated objects. If defined at an object level, we would be constrained to use inheritance in order to have the predicates available to other objects. Furthermore, this could force us to use multi-inheritance or to have some kind of generic root object containing all kinds of possible useful predicates.

For this kind of situation, Logtalk enables the programmer to encapsulate the predicates in a *category*, so that they can be used in any object. A category is a Logtalk entity, at the same level as objects and protocols. It can contain predicates directives and/or definitions. Category predicates can be imported by any object, without code duplication and without resorting to inheritance.

When defining category predicates, we need to remember that a category can be imported by more than one object. Because of that, the calls to the built-in methods that handle the private dynamic predicate (like `assertz/1` or `retract/1`) must be made using the *message to self* control structure, `::/1`. This way, we ensure that when we call one of the attribute predicates on an object, the object own definition of `attribute_/2` will be used. The predicate definitions are straightforward:

```
:- category(attributes).

      :- public(set_attribute/2).
      :- mode(set_attribute(+nonvar, +nonvar), one).

      :- public(get_attribute/2).
      :- mode(get_attribute(?nonvar, ?nonvar), zero_or_many).

      :- public(del_attribute/2).
      :- mode(del_attribute(?nonvar, ?nonvar), zero_or_many).

      :- public(del_attributes/2).
      :- mode(del_attributes(@term, @term), one).

      :- private(attribute_/2).
      :- mode(attribute_(?nonvar, ?nonvar), zero_or_many).
      :- dynamic(attribute_/2).
```

```
        set_attribute(Attribute, Value):-
              ::retractall(attribute_(Attribute, _)),
              ::assertz(attribute_(Attribute, Value)).

        get_attribute(Attribute, Value):-
              ::attribute_(Attribute, Value).

        del_attribute(Attribute, Value):-
              ::retract(attribute_(Attribute, Value)).

        del_attributes(Attribute, Value):-
              ::retractall(attribute_(Attribute, Value)).

:- end_category.
```

We have two new directives, `category/1` and `end_category/0`, that encapsulate the category code. If needed, we can put the predicate directives inside a protocol that will be implemented by the category:

```
:- category(attributes,
      implements(attributes_protocol)).

      ...

:- end_category.
```

Any protocol can be implemented by an object, a category, or both.

Importing the category

We reuse a category's predicates by importing them into an object:

```
:- object(person,
      imports(attributes)).

      ...

:- end_object.
```

After compiling and loading this object and our category, we can now try queries like:

```
| ?- person::set_attribute(name, paulo).

yes

| ?- person::set_attribute(gender, male).

yes

| ?- person::get_attribute(Attribute, Value).

Attribute = name, Value = paulo ;
Attribute = gender, Value = male ;
no
```

Summary

- Categories are similar to objects: we just put our predicate directives and definitions bracketed by opening and ending category directives.

- An object reuses a category by importing it. The imported predicates behave as if they have been defined in the object itself.

- When do we use a category instead of an object? Whenever we have a set of closely related predicates that we want to reuse in several, unrelated, objects.

A reflective class-based system

When compiling an object, Logtalk distinguishes prototypes from instance or classes by examining the object relations. If an object instantiates and/or specializes another object, then it is compiled as an instance or class, otherwise it is compiled as a prototype. A consequence of this is that, in order to work with instance or classes, we always have to define root objects for the instantiation and specialization hierarchies (however, we are not restricted to a single hierarchy). The best solution is often to define a reflective class-based system [Maes 87], where every class is also an object and, as such, an instance of some class.

In this example, we are going to define the basis for a reflective class-based system, based on an extension of the ideas presented in [Cointe 87]. This extension provides, along with root objects for the instantiation and specialization hierarchies, explicit support for abstract classes [Moura 94].

Defining the base classes

We will start by defining three classes: `object`, `abstract_class`, and `class`. The class `object` will contain all predicates common to all objects. It will be the root of the inheritance graph:

```
:- object(object,
      instantiates(class)).

      % predicates common to all objects

:- end_object.
```

The class `abstract_class` specializes `object` by adding predicates common to all classes. It will be the default metaclass for abstract classes:

```
:- object(abstract_class,
      instantiates(class),
      specializes(object)).

      % predicates common to all classes

:- end_object.
```

The class `class` specializes `abstract_class` by adding predicates common to all instantiable classes. It will be the root of the instantiation graph and the default metaclass for instantiable classes:

```
:- object(class,
      instantiates(class),
      specializes(abstract_class)).

      % predicates common to all instantiable classes

:- end_object.
```

Note that all three objects are instances of class `class`. The instantiation and specialization relationships are chosen so that each object may use the predicates defined in it and in the other two objects, with no danger of method lookup endless loops.

Summary

- An object that does not instantiates or specializes other objects is always compiled as a prototype.
- An instance must instantiate at least one object (its class). Similarly, a class must at least specialize or instantiate other object.
- The distinction between abstract classes and instantiable classes is a operational one, depending on the class inherited methods. A class is instantiable if inherits methods for creating instances. Conversely, a class is abstract if does not inherit any instance creation method.

Profiling programs

In this example, we will illustrate the use of:

- events
- monitors

by defining a simple profiler that prints the starting and ending time for processing a message sent to an object.

Messages as events

In a pure object-oriented system, all computations start by sending messages to objects. We can thus define an *event* as the sending of a message to an object. An event can then be specified by the tuple `(Object, Message, Sender)`. This definition can be refined by interpreting the sending of a message and the return of the control to the object that has sent the message as two distinct events. We call these events respectively `before` and `after`. Therefore, we end up by representing an event by the tuple `(Event, Object, Message, Sender)`. For instance, if we send the message:

```
| ?- foo::bar(X).

X = 1
yes
```

The two corresponding events will be:

```
(before, foo, bar(X), user)
(after, foo, bar(1), user)
```

Note that the second event is only generated if the message succeeds. If the message as a goal have multiple solutions, then one `after` event will be generated for each solution.

Events are automatically generated by the message sending mechanisms for each public message sent using the `::/2` operator.

Profilers as monitors

A monitor is an object that reacts whenever a spied event occurs. Two event handlers define the monitor actions: `before/3` for `before` events and `after/3` for `after` events. These predicates are automatically called by the message sending mechanisms when an event registered for the monitor occurs.

In our example, we need a way to get the current time before and after we process a message. We will assume that we have a `time` object implementing a `cpu_time/1` predicate that returns the current CPU time for the Prolog session:

```
:- object(time).

      :- public(cpu_time/1).
      :- mode(cpu_time(-number), one).

      ...

:- end_object.
```

Our profiler will be named `stop_watch`. It must define event handlers for the `before` and `after` events that will print the event description (object, message, and sender) and the current time:

```
:- object(stop_watch).

      :- uses(time).

      before(Object, Message, Sender) :-
            write(Object), write(' <-- '), writeq(Message),
            write(' from '), write(Sender), nl, write('STARTING at '),
            time::cpu_time(Seconds), write(Seconds), write(' seconds'), nl.

      after(Object, Message, Sender) :-
            write(Object), write(' <-- '), writeq(Message),
            write(' from '), write(Sender), nl, write('ENDING at '),
            time::cpu_time(Seconds), write(Seconds), write(' seconds'), nl.

:- end_object.
```

After compiling and loading the `stop_watch` object (and the objects that we want to profile), we can use the `define_events/5` built-in predicate to set up our profiler. For example, to profile all messages that are sent to the object `foo`, we need to call the goal:

```
| ?- define_events(_, foo, _, _, stop_watch).

yes
```

This call will register `stop_watch` as a monitor to all messages sent to object `foo`, for both `before` and `after` events. Note that we say "as a monitor", not "the monitor": we can have any number of monitors over the same events.

From now on, every time we sent a message to `foo`, the `stop_watch` monitor will print the starting and ending times for the message execution. For instance:

```
| ?- foo::bar(X).

foo <-- bar(X) from user
STARTING at 12.87415 seconds
foo <-- bar(1) from user
ENDING at 12.87419 seconds

X = 1
yes
```

To stop profiling the messages sent to `foo`, we use the `abolish_events/5` built-in predicate:

```
| ?- abolish_events(_, foo, _, _, stop_watch).

yes
```

This call will abolish all events defined over the object `foo` assigned to the `stop_watch` monitor.

Summary

- An event is defined as the sending of a (public) message to an object.
- There are two kinds of events: `before` events, generated before a message is processed, and `after` events, generated after the message processing completed successfully.
- Any object can be declared as a monitor to any event.
- A monitor defines event handlers, the predicates `before/3` and `after/3`, that are automatically called by the runtime engine when a spied event occurs.
- Three built-in predicates, `define_events/5`, `current_event/5`, and `abolish_events/5`, enables us define, query, and abolish both events and monitors.

Bibliography

[Alexiev 93] Alexiev, V.
Mutable Object State for Object-Oriented Logic Programming: A Survey
Technical Report TR 93-15, Department of Computing Science, University of Alberta, Canada

[Belli et al. 92]
Object-oriented programming in Prolog: rationale and a case study
Belli, F., Jack, O., Naish, L.
Technical Report 92/2, Department of Electrical and Electronics Engineering, University of Paderborn, Germany
URL: http://www.cs.mu.oz.au/~lee/papers/oolp/

[Block 89]
An Extended Frame Language
Block, F. P., Chan, N. C.
Proceedings OOPLSLA 89(10):151-157, ACM

[Bobrow 88] Bobrow, D. G., Michiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., Moon, D. A.
Common Lisp Object System Specification
ACM SIGPLAN Notices(23)

[Bratko 90] Bratko, I.
Prolog Programming for Artificial Intelligence
Addison Wesley, 2º edition, 1990

[Champaux 92] Champaux, D., Faure, P.
A comparative Study of Object-Oriented Analysis Methods
Journal of Object-Oriented Programming, Vol. 5, N.1, 1992

[Clocksin 87] Clocksin, W.F., Mellish, C.S.
Programming in Prolog
Springer-Verlag, New York

[Cointe 87] Cointe, P.
Metaclasses are First Class: the ObjVlisp Model
Proceedings OOPLSLA 87(10):156-167, ACM

[Cordes 91] Cordes, D., Brown, M.
The Literate Programming Paradigm
IEEE Computer, June 1991:52-61

[Covington 94] Covington, M. A.
ISO Prolog: A Summary of the Draft Proposed Standard
URL: ftp://ai.uga.edu/pub/prolog.standard/

[Cox 86] Cox, Brad J.
Object-Oriented Programming: An Evolutionary Approach
Addison-Wesley Publishing Company, Don Mills, Ontario

[Davison 89] Davison, A.
Polka: A Parlog Object oriented language
Ph.D. Thesis, Imperial College, London

[Davison 92] Davison, A.
A survey of logic programming-based object oriented languages
Tech Report 92/3, Dept. of Computer Science, University of Melbourne, Australia
URL: http://www.cs.mu.oz.au/tr_db/mu_92_03.ps.gz

[Davison 93] Davison, A.
The deductive and object oriented features of BeBOP
Tech Report 93/6, Dept. of Computer Science, University of Melbourne, Australia
URL:http://www.cs.mu.oz.au/tr_db/mu_93_06.ps.gz

[Delzanno 97] Delzanno, G.
Logic and Object-Oriented Programming in Linear Logic
Ph.D. Thesis, University of Pisa, Italy
URL:http://www.mpi-sb.mpg.de/~delzanno/

[Dony 90] Dony, C.
Exception Handling and Object-Oriented Programming: Towards a Synthesis
Proceedings OOPLSLA 90:322-330, ACM

[Fornarino 89] Fornarino, M., Pinna, A.-M.,Trousse, B.
An Original Object-Oriented Approach for Relation Management
Proceedings of the 4th Portuguese Conference on Artificial Intelligence
Lecture Notes in Artificial Intelligence, Springer-Verlag (390):13-26

[Fromherz 93] Fromherz, M.
OL(P): Object Layer for Prolog
URL: ftp://parcftp.xerox.com/ftp/pub/ol/

[Fukunaga 86] Fukunaga, K., Hirose, S.
An Experience with a Prolog-based Object-Oriented Language
Proceedings OOPLSLA 86, 21(11):224-231, ACM

[Goldberg 83] Goldberg, A., Robson, D.
Smalltalk-80 The language and its implementation
Addison-Wesley Series in Computer Science

[Gosling et al. 96] Gosling, J., Joy, B., Steele, G.
The Java Language Specification
Addison-Wesley, Reading, MA

[ISO 95] Joint Technical Committee ISO/IEC JTC 1
ISO/IEC DIS 13211-1 - Programming Language Prolog Part 1: General Core
URL: http://www.iso.ch/cate/d21413.html

[Knuth 84] Knuth, D. E.
Literate Programming
Computer Journal, May 84, 27(2):97-111

[Lieberman 86] Lieberman, H.
Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems
Proceedings OOPLSLA 86:189-214, ACM

[Maes 87] Maes, P.
Concepts and Experiments in Computational Reflection
Proceedings OOPLSLA 87, ACM

[McCabe 92] McCabe, F. G.
Logic and Objects
Prentice Hall Series in Computer Science

[Moon 86] Moon, D.
Object-Oriented Programming in Flavors
Proceedings OOPLSLA 86:1-8, ACM

[Moss 94] Moss, C.
Prolog++ The Power of Object-Oriented and Logic Programming
Addison-Wesley International Series in Logic Programming

[Moura 94] Moura, P., Costa, E.
Logtalk: Programação Orientada para Objectos em Prolog
2ª Conferência e Exposição Portuguesa de Tecnologia Orientada por Objectos
3i Consultores, Liboa

[Moura 99] Moura, P.
Porting Prolog: Notes on porting a Prolog program to 22 Prolog compilers or the relevance of the ISO Prolog standard
ALP Newsletter, 12/2, May 1999

[Razek 92] Razek, G.
Combining Objects and Relations
Comunications of the ACM, 27(12):66-70

[Rumbaugh 87] Rumbaugh, J.
Relations as Semantic Constructs in an Object-Oriented Language
Proceedings OOPLSLA 87:466-481, ACM

[Rumbaugh 88] Rumbaugh, J.
Controlling Propagation of Operations using Attributes on Relations
Proceedings OOPLSLA 88:285-296, ACM

[Schachte 95] Schachte, P., Saab, G.
Efficient Object-Oriented Programming in Prolog
Logic Programming: Formal Methods and Pratical Applications
Studies in Computer Science and Artificial Intelligence, 11
Elsevier Science B.V. North-Holland, Amsterdam, 1995

[SICStus 95] SICStus
SICStus Prolog Manual
URL: http://www.sics.se/ps/sicstus.html

[Shan 93] Shan, Y., Cargill, T., Cox, B., Cook, W., Loomis, M., Snyder, A.
Is Multiple Inheritance Essential to OOP? (Panel)
Proceedings OOPLSLA 93:360-363

[Stefik 86] Stefik, M. J., Bobrow, D. G. , Kahn, K. M.
Integrating Acess-Oriented Programming into a Multiparadigm Environment
IEEE Software, January 1986:10-18

[Stroustrup 86] Stroustrup, B.
The C++ Programming Language
Addison-Wesley Series in Computer Science

[Taenzer 89] Taenzer, D., Ganti, M., Podar, S.
Problems in Object-Oriented Software Reuse
Proceedings of ECOOP 89
British Computer Society Workshop Series, Cambridge University Press

[Tanzer 95] Tanzer, C.
Remarks on Object-Oriented Modeling of Associations

Journal of Object-Oriented Programming, February 1995, SIGS Publications

[Tanenbaum 87] Tanenbaum, A.
Operating Systems - Design and Implementation
Prentice-Hall Software Series

[Welsch 89] Welsch, C., Barth, G.
Reasoning Objects with Dynamic Knowledge Bases
Proceedings of the 4th Portuguese Conference on Artificial Intelligence (390):257-268
Lecture Notes in Artificial Intelligence, Springer-Verlag

Glossary

ancestor
: A class or parent that contributes to the definition of an object. The ancestors of an object are its class and all the superclasses of its class or its parent and the ancestors of its parent.

category
: A set of predicates directives and clauses that can be imported by any object.

class
: An object that defines the common predicates of a set of objects (its instances).

  abstract class
  : A class that can not be instantiated. Usually used to store common predicates that are inherited by other classes.

  metaclass
  : The class of a class, when we see it as an object. Metaclass instances are themselves classes. In a reflexive system any metaclass is also an object.

  subclass
  : A class that is a specialization, direct or indirectly, of another class.

  superclass
  : A class from each another class is a specialization (direct or indirectly, via another class).

directive
: A term that affects the compilation and use of Prolog code. Directives are represented using the operator `:-/1` as a prefix functor.

  entity directive
  : A directive that affects how Logtalk entities (objects, protocols, or categories) are used or compiled.

  predicate directive
  : A directive that affects how predicates are called or compiled.

encapsulation
: The hiding of an object implementation. This promotes software reuse by isolating users from implementation details.

entity
: Generic name for Logtalk compilation units: objects, categories and protocols.

event
: The sending of a message to an object. An event can be expressed as an ordered tuple: `(Event, Object, Message, Sender)`. Logtalk distinguish between the sending of a message — `before` event — and the return of control to the sender — `after` event.

identity
: Property of an entity that distinguish it from every other entity. In Logtalk an entity identity can be an atom or a compound term. All Logtalk entities, objects, protocols and categories share the same name space.

inheritance
: An object inherits predicate directives and clauses from other objects that it extends or specializes. If an object extends other object then we have a prototype-based inheritance. If an object specializes or instantiates another object we have a class-based inheritance.

  private inheritance
  : All public and protected predicates are inherited as private predicates.

  protected inheritance
  : All public predicates are inherited as protected. No change for protected or private predicates.

  public inheritance
  : All inherited predicates maintain the declared scope.

instance
: The same as object. This term is used when we want to emphasize that an object characteristics are defined by another object (its class).

instantiation
: The process of making a new class instance.

message
: A request for a service, sent to an object. In more logical terms, a message can be seen as a request for proof construction using an object's predicates.

metainterpreter
: A program capable of running and modifying other programs written in the same language.

method
: Set of clauses used to answer a message sent to an object. Logtalk uses dynamic binding to find which method to run to answer a message.

monitor
: Any object that is notified when a spied event occurs. The spied events can be set by the monitor or by any other object.

object
: An entity characterized by an identity and a set of predicate directives and clauses. In Logtalk objects can be either static or dynamic, like any other Prolog code.

  parametric object
  : An object whose name is a compound term containing free variables that can be used to parameterize the object predicates.

parent
: An object that is extended by another object.

predicate
: Predicates describe what is true about the application domain. A predicate is identified by its name and number of arguments.

  local predicate
  : A predicate that is defined in an object (or in a category) but that is not listed in a scope directive. These predicates behave like private predicates but are invisible to the reflection methods.

metapredicate
: A predicate where one of its arguments will be called as a goal. For instance, `findall/3` and `call/1` are Prolog built-ins metapredicates.

private predicate
: A predicate that can only be called from the object that contains the scope directive.

protected predicate
: A predicate that can only be called from the object containing the scope directive or from an object that inherits the predicate.

public predicate
: A predicate that can be called from any object.

visible predicate
: A predicate that is declared for an object, a built-in method or a Prolog or Logtalk built-in predicate.

profiler
: A program that collects data about other program performance.

protocol
: A set of predicates directives that can be implemented by an object or a category (or extended by another protocol).

prototype
: A self-describing object that may extend or be extended by other objects.

self
: The original object that received the message under execution.

sender
: An object that sends a message to other object.

specialization
: A class is specialized by constructing a new class that inherit its predicates and possibly add new ones.

this
: The object that contains the predicate clause under execution.

Index